Section 24
Parser Function

The *Parser* function of the ETMS is responsible for getting message packets of National Airspace System (NAS) data and Flight Schedule data from the *NAS Data Distributor (NAS.DIST)*. The relevant data is extracted, converted, and passed to the *Flight Database Processor* (*FDB*). The most notable data conversion is the transformation of a flight path into an event list.

Processing Overview

The *Parser* function consists of five processes (*queue_driver*, *feedback.receiver*, *parser.relay*, *routedb.relay* and *parser*), as shown in Figure 24-1. The *queue_driver* process reads packets of data from the *NAS Data Distributor* and enqueues valid data packets for the *Parser* process. The *queue_driver* process communicates with the *NAS.DIST* via a network addressing connection and with the *Parser* process via a queue. The *Parser* process extracts and converts relevant data parsed from the input data packets and passes the data to the *parser.relay* process. The *Parser* process communicates with the *parser.relay* process via another queue. The *parser.relay* process sends parsed data to the fdb receiver which places the data into a queue for the *Flight Database Processor*. The *Parser* process also forwards parsed flight plan (FZ) data to the *routedb.relay* process via a queue. The *routedb.relay* process sends parsed FZ data to the *routedb.receiver* process. The *feedback.receiver* process transfers incomplete flight amendment (AF) messages from the feedback relay process of the *Freight Database Processor* back to the *Parser* process for more complete data transformation. Most AF messages specify a new flight path (field 10) which then has to be converted by the *Parser* process into an event list. An altitude and speed are necessary for this conversion but the AF message rarely contains all this information. The **flight database** has this information stored from previous messages for this particular flight, so when an incomplete AF message gets to the *Flight Database Processor*, the stored speed and altitude are filled into the data structure and passed back to the *Parser* process, via the *feedback.receiver* process. The *Flight Database Processor* communicates with the *Parser* process via a mailbox, created by the *feedback.receiver* process.

Because running in real time is an important consideration, it became necessary to cut down on costly input/output (I/O) wait time. Therefore, the queueing method of inter-process communications was developed in order to separate processing time from I/O wait time, as well as to hold data temporarily in case of a backlog (see Section 23 for more information on queues).

24.1 The queue_driver Process

Purpose

The *queue_driver* process is the first process within the *Parser* function to come into play. Its purpose is to hook up to the *NAS.DIST*, read data blocks from the service provider, and enqueue valid data blocks, which are to be dequeued and parsed by the *Parser* process.

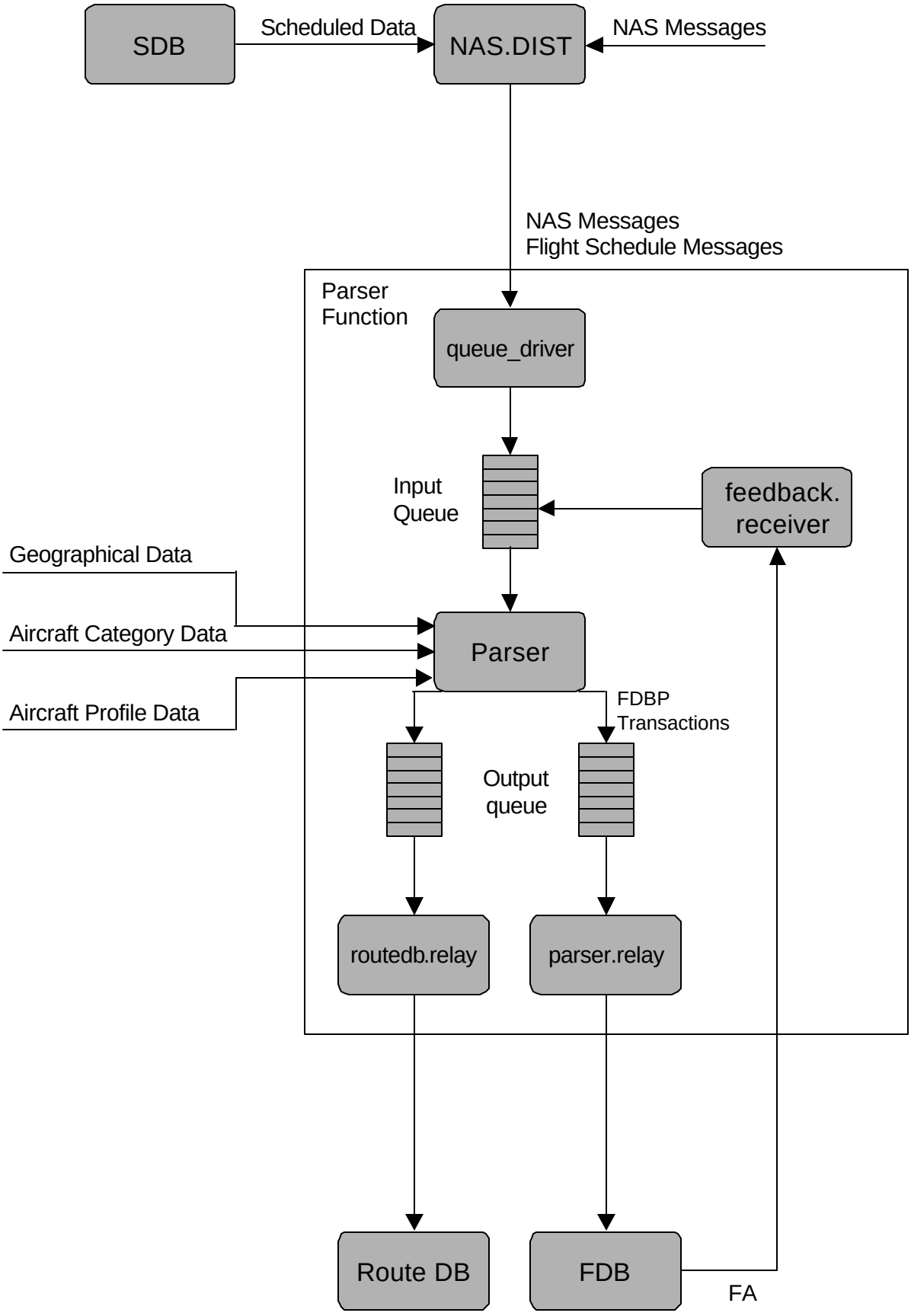


Figure 24-1. Data Flow of the Parser Function

Execution Control

The *Parser* function was designed to be manually or automatically started (or restarted) if problems occur. Most often, restarting is done automatically via the *Nodescan* process that runs on the same node as the *Parser* function (see Section 33 on *Nodescan*).

The *queue_driver* process is started automatically by *Nodescan* or manually by an ETMS operator. If problems occur, the *queue_driver* can be restarted without bringing any of the other *Parser* function processes down. This ensures a continuous flow of parsed data even if other processes are having trouble. If the *queue_driver* process crashes, all data stored in queues can be recovered when it is restarted.

Input

The *queue_driver* input is comprised of the following four types of separate messages, grouped into packets of messages, called *subtypes*, by the *NAS.DIST*:

(1) NAS messages, consisting of the following subtypes:

   (a) amendment (AF)

   (b) arrival (AZ)

   (c) departure (DZ)

   (d) flight plan (FZ)

   (e) cancellation (RZ)

   (f) position update (TZ)

   (g) boundary crossing (UZ)

   (h) oceanic position updates (TO)

   (i) ARTCC status (CCC)

(2) Schedule Data messages, consisting of the following subtypes:

   (a) scheduled flight plan (FS)

   (b) scheduled cancellation (RS)

(3) Test messages, which are used for checking if local network connections are working properly

(4) Flight path data from the *Flight Database Processor* (FA).

The input message packet is an array (of up to 8192 characters) filled with as many separate messages as can fit without overflowing the array's bounds. Within the array, the messages are separated from one another by a linefeed. (FA messages are individually stored in 8192-byte buffers.)

Output

The *queue_driver* process passes to the *Parser* process packets of NAS and Flight Schedule messages, previously listed under the *Input* subheading.

Processing

The first action of the *queue_driver* process, as illustrated in Figure 24-2, is to create a time stamped pad for all output. This pad is closed daily so an output pad does not lock up too much disk space.

The *queue_driver* process then creates the stack and the queues needed for inter-process communications within the entire *Parser* function. The stack is used to keep track of available memory information (see Section 15 on queues). Three queues need to be created: one between the *queue_driver* process and the *Parser* process, one between the *Parser* process and the *parser.relay* process, and one between the *Parser* and the *routedb.relay* process. The queueing technique has been designed so that if a process crashes and has to be restarted, that process will use the previously created stack and queues to hook up to the next process. This ensures that any data in a queue at the time of a crash will still be there when the process is restarted.

The *queue_driver* process then creates the *feedback.receiver* as a child process. This receiver allows communications from the *Flight Database Processor* to the *Parser* process.

The *queue_driver* process then does all necessary network addressing initialization, builds the service provider address to search the network for NAS.DIST, and attempts to register with NAS.DIST. The service provider address contains the site switch name passed in on the command line, NAS.DIST class name, any node switch, any invocation number and '0' subaddress. If the registration is unsuccessful, the *queue_driver* will periodically try to reregister. (Up to 15 consecutive times; otherwise terminate writing for restart.)

Due to the asynchronous nature of network addressing, it is possible to get NAS data messages while the *queue_driver* is waiting for a registration response. The *queue_driver* processes such messages and enqueues them for the *parser* process.

Once the *queue_driver* process has registered with the *NAS.DIST*, it reads message packets from the service provider. As a packet is read from the network addressing port, the *queue_driver* process checks that the port is functioning properly by looking at the status of the completed **read** operation. If at any time there is a problem with the network addressing connection (network or mailbox crash), the queue driver will try to reestablish a connection.

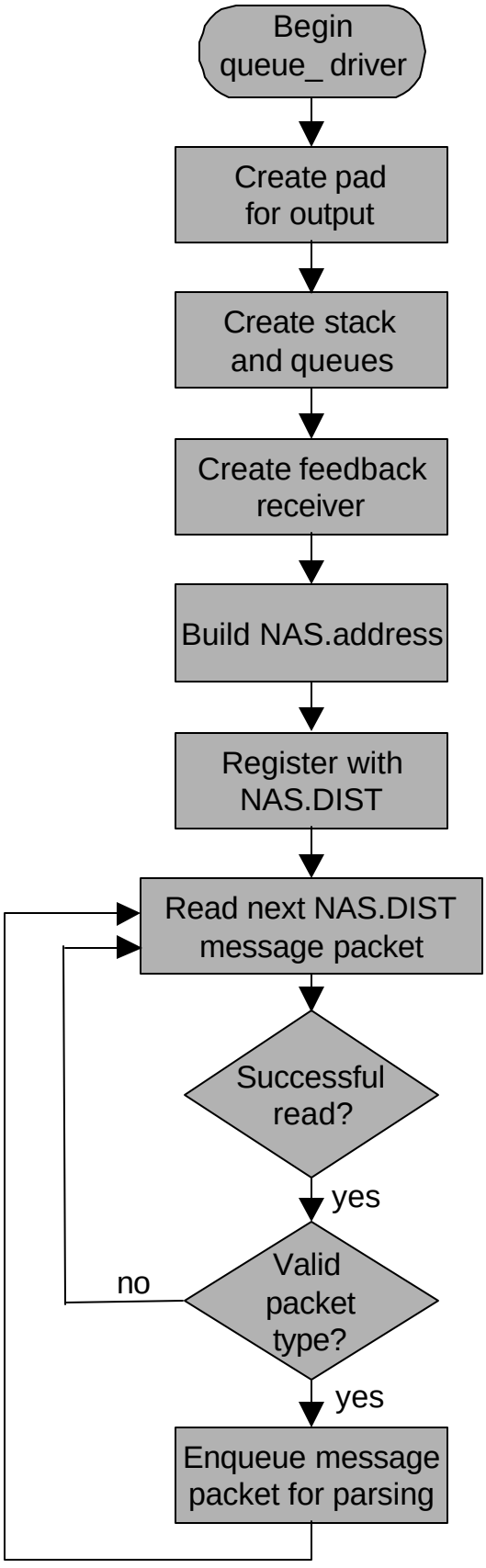


Figure 24-2. Sequential Logic of queue_driver Process

Once a message packet has been validated, the *queue_driver* process enqueues it so that it can be dequeued and parsed by the *Parser* process. The *queue_driver* process then waits for the next message packet to come in through the *NAS.DIST*, at which time the entire sequence is repeated.

Error Conditions and Handling

Any errors that occur during the creation of the output pad, stack and queues, *feedback.receiver* process or mailbox operations are reported to the screen along with a diagnostic message. If the error is terminal, the program exits and is restarted by nodescan.

24.2 The feedback.receiver Process

Purpose

The *feedback.receiver* process is set up to provide communications from the *Flight Database Processor* to the *Parser* process. It is used to pass necessary data previously stored in the **flight database** back to the *Parser* process to aid in route conversion. Since queues have to be set up on local nodes, a method of enqueuing a message packet on one node and dequeuing that message packet on another node has been devised.

Execution_control

The *queue_driver* process invokes the *feedback.receiver* process as a standalone process.

Input

A data structure holding all information for a particular flight stored in the **flight database**.

Output

The same data structure, enqueued in the *Parser* process input queue.

Processing

The *feedback.receiver* process creates a socket and hooks into the existing input queue between the *queue_driver* process and the *Parser* process. The *Flight Database Processor* connects to this socket, and the feedback.receiver passes completed flight data (specified as an FA message) into the input queue for the *Parser* process.

Error Conditions and Handling

Any errors which occur during the operation of the *feedback.receiver* process are reported to the screen. Any terminal error is reported, and the process crashes.

24.3 The parser.relay Process

Purpose

The *parser.relay* process is set up to aid queue operations between different processes. It is used to enqueue parsed data packets to be dequeued by the *Flight Database Processor*. The *parser.relay* process takes data that has been enqueued by the *Parser* process and sends it via a socket to the *Flight Database Processor*.

Execution_control

The *Parser* process invokes the *parser.relay* process as a stand-alone process. It is automatically restarted if any problems occur (see Section 15 on relays).

Input

Enqueued *Flight Database Processor* transactions. Refer to section 24.7 for a list of the transaction types and information regarding them.

Output

Enqueued *Flight Database Processor* transactions. Refer to section 24.7 for a list of the transaction types and information regarding them.

Processing

The *Parser* process, once all data conversions for a single NAS message have been completed, enqueues a data structure into the *parser.relay* process queue (set up by the *queue_driver* process). The *parser.relay* process dequeues this data and ships it through a socket (set up by the *Flight Database Processor*) to the *Flight Database Processor*.

Error Conditions and Handling

Any errors which occur during the operation of the *parser.relay* process are reported to the screen. If a terminal error is reported, the process exits.

24.4 The routedb.relay Process

Purpose

The *routedb.relay* process is set up to aid queue operations between different processes. It is used to enqueue parsed data packets to be dequeued by the *Route Database Processor.* The *routedb.relay* process takes data that has been enqueued by the *Parser* process and sends it via a socket to the *Route Database Processor.*

Execution_control

The *Parser* process invokes the *routedb.relay* process as a stand-alone process. It is automatically restarted if any problems occur (see Section 15 on relays).

Input

Enqueued *Route Database Processor* transactions. Refer to section 24.7 for a list of the transaction types and information regarding them.

Output

Enqueued *Route Database Processor* transactions. Refer to section 24.7 for a list of the transaction types and information regarding them.

Processing

The *Parser* process, once all data conversions for a single NAS message have been completed, enqueues a data structure into the *routedb.relay* process queue (set up by the *queue_driver* process). The *routedb.relay* process dequeues this data and ships it through a socket (set up by the *Route Database Processor*) to the *Route Database Processor*.

Error Conditions and Handling

Any errors which occur during the operation of the *routedb.relay* process are reported to the screen. If a terminal error is reported, the process exits.

24.5 The Parser Process

Purpose

The purpose of the *Parser* process is to break down a NAS, OMP, or Flight Schedule message into its constituent parts, convert data to an internal form, and pass the parsed data to the *Flight Database Processor*. The *Parser* process dequeues packets of NAS or Flight Schedule messages and separates them into single messages. A single NAS or Flight Schedule message then has its separate fields extracted, converted if necessary, and stored. The extracted data is sent to the *Flight Database Processor* once the entire message has been successfully parsed.

Execution Control

The *Parser* process is started automatically by *Nodescan* or manually by an ETMS operator. If problems occur, it can be restarted without bringing any of the other *Parser* function processes down. This ensures a continuous flow of parsed data even if other processes are having trouble. If the *Parser* process crashes, all data stored in queues can be recovered when it is restarted.

Input

Following is the *Parser* process input data:

(1) NAS messages — messages containing flight data to be parsed

(2) OMP messages — messages containing oceanic position data

(3) Flight Schedule messages — messages containing scheduled flight data from the OAG to be parsed

(4) Feedback messages — messages returned to the Parser for processing by the FDB and to control message flow to the FDB

(5) Geographical Data — numeric boundaries (latitude/longitude) for Air Route Traffic Control Centers (ARTCCs), sectors, fixes, etc.

(6) Aircraft Category Data — category values (weight, military, etc.) for a given aircraft type

(7) Aircraft Profile Data — performance characteristics for a given aircraft type used by the *Flight Profile Modeling* module to model an aircraft flight

(8) National Route Program (NRP) parameter data, containing information to be used for validating field 3 and field 11 portions of NAS messages.

Output

The *Parser* process enqueues converted data (extracted from each input message) to the *parser.relay* process. The *Parser* also enqueues converted flight plan data to the *routedb.relay* process.

Processing Overview

There are five main tasks associated with the *Parser* process, as depicted in Figure 24-3. The first task is the initialization of the system. The second task entails the parsing of input messages and converting the data. The third task is the most notable data conversion, processing a flight path into an event list. The fourth task is the modeling of the aircraft flight. The fifth and final task entails the cleaning up of the event list. All of these tasks are performed by individual *Parser* process modules.

The *Parser Initialization* module initializes all values for the *Parser* process prior to the parsing of any input messages (see Section 24.4.1).

The *Parse Messages* module extracts data from the input messages and does some minor data conversions (see Section 24.4.2).

The *Route Processor* module is called to translate a flight path into an event list (see Section 24.4.3). The *Parser* process provides all the data necessary to make this conversion (see Table 24-1). Event lists are created only from messages that contain route fields (i.e., FS, FZ, UZ, FA, and a few AF messages).

The *Flight Profile Modeling (assign_a_profile)* module supplies aircraft characteristics to the *Route Processor* module so it can accurately model the aircraft's route of flight (see Section 24.4.4.2). The *assign_a_profile* module uses specific aircraft category data, which depend on the type of aircraft being used, to model the aircraft's performance.

If the *Route Processor* module was able to create an event list from the given data, the *cleanup_evlist* module checks the event list for any idiosyncrasies and resolves them (see Section 24.4.5).

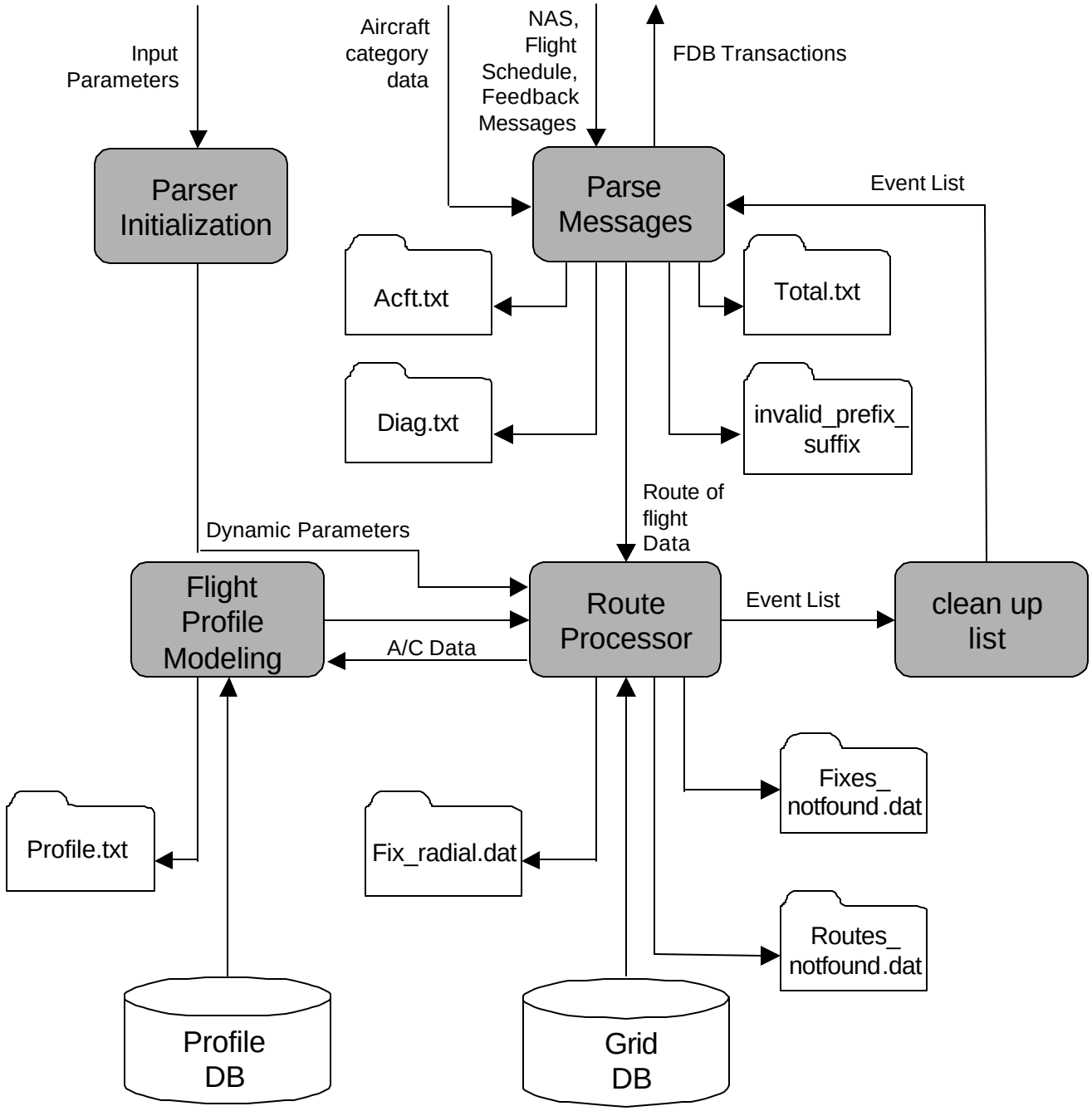


Figure 24-3. Data Flow of the NAS Message Parser Process

Table 24-1. erect Data Structures

erect (event record type)

Library Name: ttm_openlib	**Element Name:** event.h

Purpose: To contain information about an event. The contents and type of each data item is shown here.

Field Name: time_index **Field Type:** INT32

Data Item	Definition	Unit/Range	Which Bits?
event kind	Is this event an arrival or departure?	0 - 2	31 - 30
phase	In which phase of the flight does the event occur?	enumerated type from TAKEOFF to LANDING (0-6)	29 - 27
time	At what time does this event occur?	minutes from midnight	26 - 15
TDB binary time_type	Is this an actual or predicted event?	1 means actual, 0 means predicted	14
time type	Desc. the type of the event time (actual, predicted).	constants defining time types range from 0 - 7	13 - 11
unused	--	--	10 - 0

Field Name: del_alt-vel **Field Type:** INT32

Data Item	Definition	Unit/Range	Which Bits?
delay	Any filed airborne delay.	minutes	31 – 22
altitude	The altitude of this flight at this event.	flight level in hundreds of feet	21 – 12
velocity	The velocity of this flight at this event.	nautical miles per minute	11-0

Field Name: distance **Field Type:** short

Data Item	Definition	Unit/Range	Which Bits?
waypoint flag	Is this position of this event a waypoint of the flight?	1 means yes, 0 no	15
unused	--	--	14
distance	The distance this flight has flown from the last event.	nautical miles	13 - 0

Table 24-1. erect Data Structures (continued)

erect (event record type)			
Field Name: heading_type		**Field Type:** short	
Data Item	**Definition**	**Unit/Range**	**Which Bits?**
monitor flag	Is the element of this event monitored?	1 means yes, o no	15
heading	The heading of this flight at this event.	0 – 359 degrees	14 – 6
element type	At what type of element does this event occur?	These types currently range from 0 to 18.	5 - 0
Field Name: element_indes		**Field Type:** short	
Data Item	**Definition**	**Unit/Range**	**Which Bits?**
element_indes	The element's index in the grid database/	0 – 65535	15 - 0
Field Name: latitude		**Field Type:** short	
Data Item	**Definition**	**Unit/Range**	**Which Bits?**
latitude	The latitude of the position of this event.	radians times 1000	15 - 0
Field Name: longitude		**Field Type:** short	
Data Item	**Definition**	**Unit/Range**	**Which Bits?**
longitude	The longitude of the position of this event.	radians times 1000	15 - 0

24.5.1 The Parser Initialization Module

Purpose

The *Parser Initialization* module sets up the initial state of the *Parser* process. Once everything is properly initialized, parsing of input messages may begin.

Input

Parameters consisting of necessary input files and dynamic variables used throughout the *Parser* process.

Output

Dynamic parameters read from input files and passed to the *Route Processor* module specifying which types of actions to carry out.

Processing

The *Parser Initialization* module (as shown in Figure 24-4) starts off by creating a time stamped pad for all output. This pad is closed daily so an output pad does not occupy too much disk space.

The next step is initializing all elements to be used throughout the *Parser* process. This task encompasses acquiring memory, initializing variables and data structures, and initializing the **grid database** (see Section 19).

The *Parser Initialization* module then maps into virtual memory certain previously created aircraft category files that hold data for all known aircraft types. These files are used throughout the *Parser* process.

The *Parser Initialization* module then reads the local clock time and computes a current time stamp, which consists of month, day, hour, and minute. All files opened by the *Parser Initialization* module have this time stamp appended to their pathnames. The *Parser Initialization* module then passes the time stamp to the *Route Processor* module so it can append the same time stamp to its files. Files are closed and new files opened (with current time stamps) every twenty-four hours. This allows the user to save, scan through, or delete previous files without having to worry about whether the file is currently locked by the *Parser* process.

The *Parser Initialization* module's next action involves the opening of static data files used by the *assign_a_profile* module. Since the *assign_a_profile* module doesn't stand on its own but is part of the *Parser* process, the *Parser Initialization* module is responsible for opening certain mapfiles needed by the *assign_a_profile* module. Pointers to these files are used by the *assign_a_profile* module, allowing access to the desired data.

The next step is to read the NRP parameter files for **field 3** and **field 11** parsing.

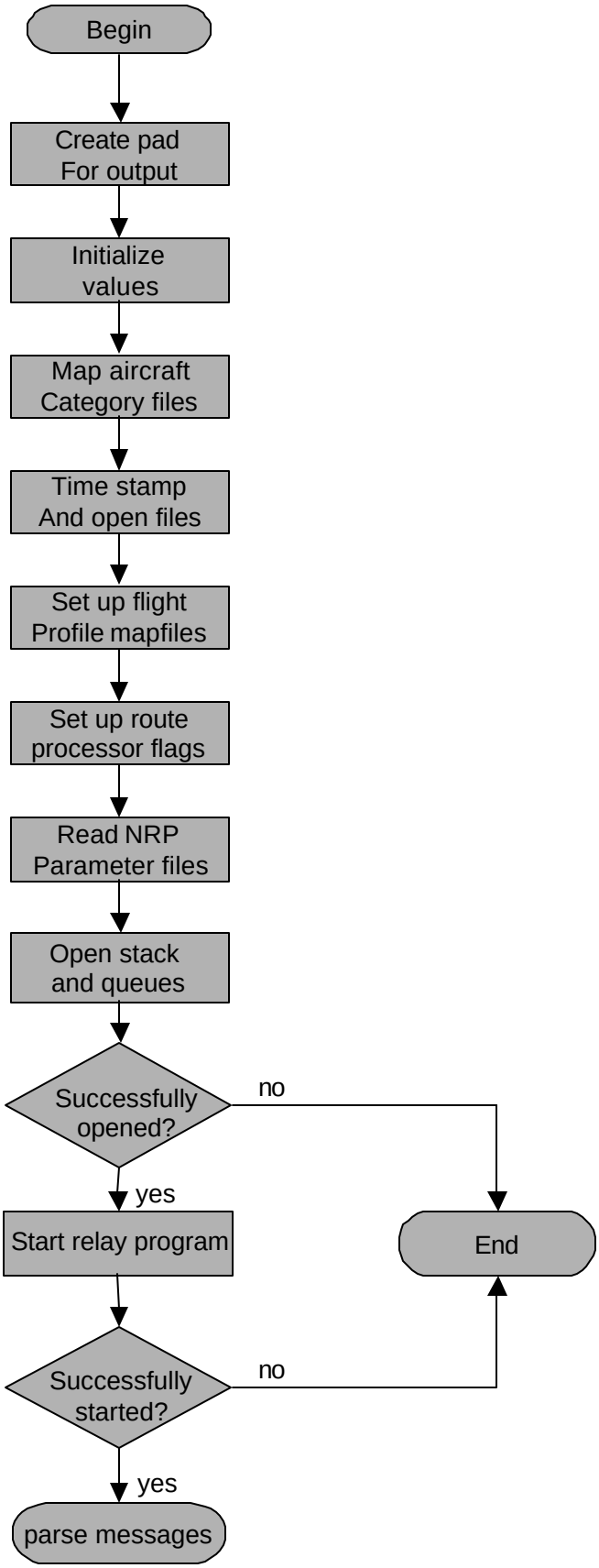


Figure 24-4. Sequential Logic for Parser Initialization Module

The *Parser Initialization* module reads values which are used as flags by the *Route Processor* module. The flags primarily enable/disable the printing of certain debugging information to the screen from the *Route Processor* module.

The *Parser Initialization* module then opens the stack and queues needed for data communication. The stack and queues are created by the *queue_driver* process and, once opened, allow access to the NAS or Flight Schedule message packets that the *queue_driver* process or feedback.receiver has enqueued. If there is a problem opening the stack and queues, the *Parser* process outputs diagnostic messages and exits, so it can get restarted by *Nodescan.*

Once the stack and queues have been opened successfully, the *parser.relay* and *routedb.relay* processes are invoked. If there is a problem invoking either process, the *Parser* process exits, to get restarted by *Nodescan*.

Error Conditions and Handling

Any errors that occur in the *Parser Initialization* module are logged in a diagnostic error file. Any terminal error is logged and the process terminates, waiting for the *Nodescan* process to restart the *Parser* process.

24.5.2 The Parse Messages Module

Purpose

The *Parse Messages* module separates input message packets into individual messages, identifies the message types, and extracts the necessary data from each message.

Input

Following is the *Parse Messages* module input data:

(1) Raw flight messages — NAS, Flight Schedule, and OMP messages

(2) Feedback messages — complete data for a requested flight

(3) Aircraft category data — values for all known aircraft types

(4) Completed event list —returned from the *cleanup_evlist* module

Output

Following is the *Parse Messages* module output data:

(1) Route of flight data — values needed for the *Route Processor* module to create an event list

(2) FDB transactions — parsed data to be stored in the **flight database**

Processing

The *Parse Messages* module (as shown in Figure 24-5) starts off by dequeueing a packet of data. The data packet consists of input messages separated by linefeeds.

Parse Messages extracts a single input message from the packet and identifies its type. Once the message type is known, the *Parse Messages* module calls the appropriate routine to extract the data from that message type.

Since every NAS message type has a structured format (see Section 6 on NAS messages), the data may be easily extracted in most cases. Some fields may hold different possible data formats (e.g., a coordination fix may be from two to twelve characters long) or different field types (e.g., an FZ message may have in its seventh field either an assigned or a requested altitude format). If the formats match when checked, the *Parse Messages* module extracts the data, which are converted if necessary (e.g., speed is input in knots but gets converted to [nautical miles per minute] * 100) and then stored.

When the *Parse Messages* module has successfully extracted all the data, the data is enqueued and the *parser.relay* process sends the data to the *Flight Database Processor.* If the data is based on a flight plan (FZ), then the data is also enqueued to the *routedb.relay* process. However, not all messages are parsed successfully. Occasionally, data extraction errors occur, usually due to a malformed message. If these errors are perceived to be minor, all correctly parsed data are sent to the *Flight Database Processor*. If a serious error occurs (e.g., no correct fields were found), a message is catalogued in an error file, and the *Parse Messages* module discards the data instead of sending the data to the *Flight Database Processor*.

Sending data to the *Flight Database Processor* simply consists of placing the data in a previously created queue. Once in the queue, this data is dequeued by the parser.relay process, transferred via a socket, and placed into a queue local to the Flight Database Processor processing node. To reduce network loading, the Parser only places the data into the queue when the Parser is on the "Master" processing string. The FDB informs the Parser of its status (master or slave) by periodically sending a message to the parser via the feedback. receiver.

Once the *Parse Messages* module has singled out an individual message from a data packet, extracted and (when necessary) converted the data from that message, and passed the extracted data to the *Flight Database Processor*, it has completed one cycle of its process. The next action of the *Parse Messages* module is to single out the next input message from the message packet and repeat the extraction sequence. If there are no more input messages in the message packet, then the *Parse Messages* module dequeues the next message packet and repeats the entire data extraction process.

To give an example of how data from an input message are extracted, the bottom line parsing routine *fill_struct_fz* will be examined (see Figure 24-6). This is a low-level routine which parses the data from a flight plan (FZ) message. It calls basic string manipulation routines which read characters from the current spot in the message until either a certain number of characters are read or a specific delimiter is encountered. The number of characters and a set of delimiters are passed as parameters to the string manipulation routines. All message types have a similar parsing routine associated with their format.

The first action of the *fill_struct_fz* routine is to clear local variables that are to be used during the parsing of the message. The encoded time stamp, the ARTCC of origin, and the message type have been previously parsed, leaving the first unique field to be parsed.

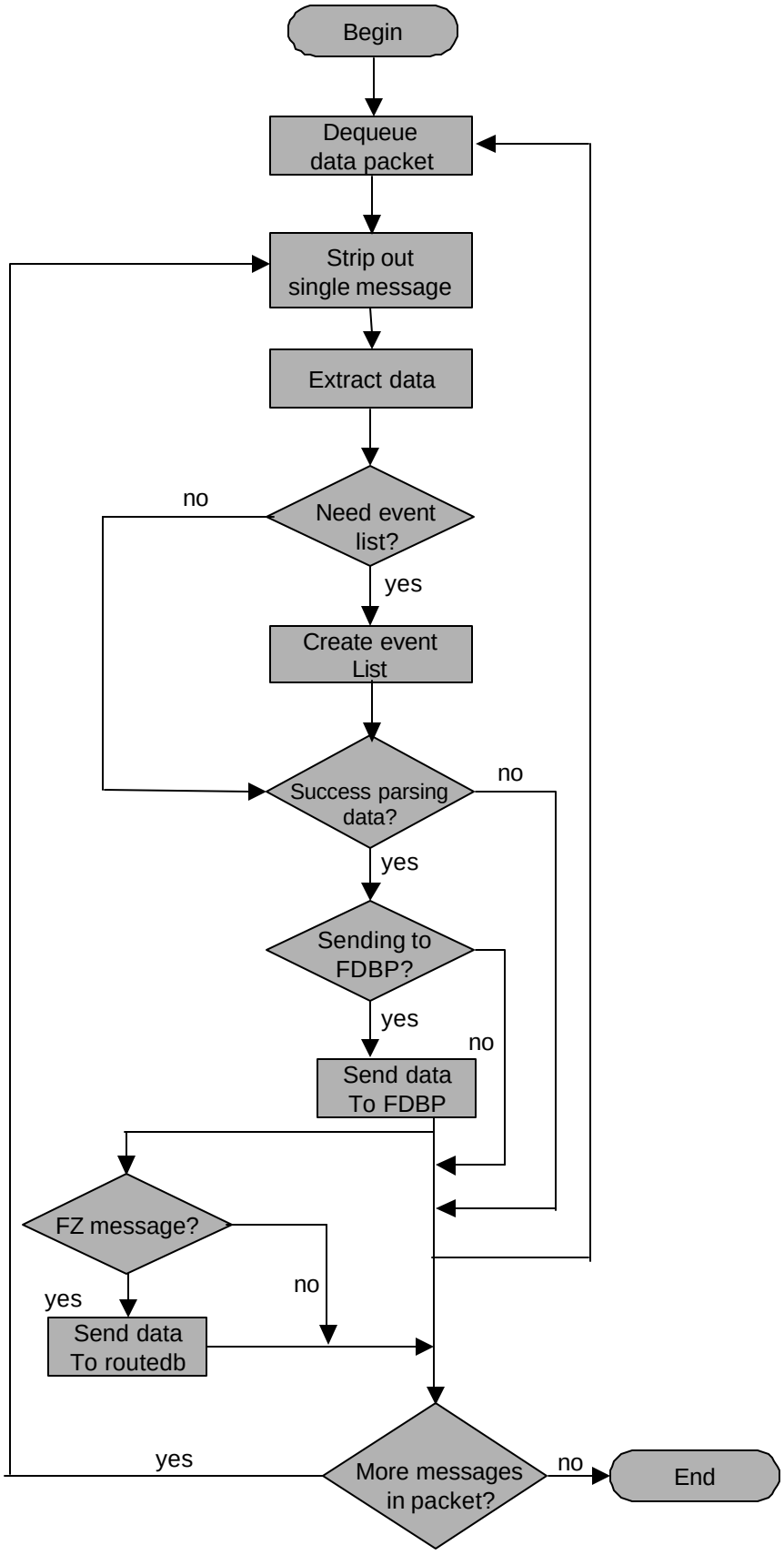


Figure 24-5. Sequential Logic for Parser Messages Module

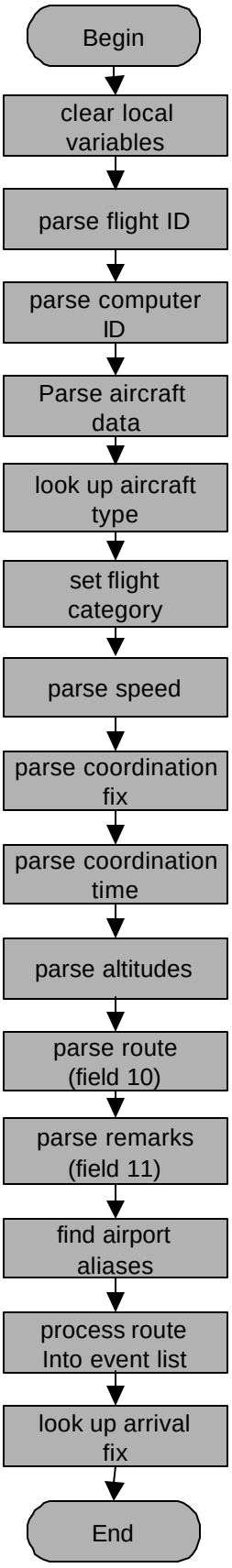


Figure 24-6. Sequential Logic for the fill_struct_fz Routine

The first field to be parsed contains the flight ID. Characters are processed until either seven characters have been read (the current maximum size for a flight ID field), or a backslash or a space is encountered.

The next field to be parsed contains the (optional) computer ID. Characters are read until a space is encountered. The field is then checked for proper syntax. If the field was not comprised of exactly three numeric characters, then the routine assumes that the optional computer ID was not specified and that the field just parsed is the next required field.

The next field to be parsed consists of aircraft data. Characters are read until a space is encountered. As many as three elements, separated by the "/" character, may be contained in this field. The mandatory field, the aircraft type, may be accompanied by a preceding equipment prefix identifier and a succeeding equipment suffix. The equipment prefix identifier may have one or two characters, while the equipment suffix identifier contains one character. The optional fields are validated against values specified in the NRP parameters files. Once the aircraft type has been read, associated data for that type are looked up in the aircraft category map file. The flight category is then set, depending on the flight ID and values returned from the aircraft type lookup. These values are saved in the same data structure as the other parsed fields, later to be passed to the *Flight Database Processor* for storage.

The aircraft speed is the next field to be parsed. Characters are read until a space is encountered. The speed syntax is checked to make sure it is numeric. It is then converted to an internal format (nautical miles per minute * 100).

The next field to be parsed is the coordination fix field. Characters are read until a space is encountered. This field can consist of an airport name, a fix name, or a latitude/longitude value.

Coordination time is the next field to be parsed. Characters are read until a space is encountered. This value is checked to be numeric and is then converted to minutes after midnight.

The altitude field is parsed next. Characters are read until a space is encountered. This value is also checked to be numeric.

The next field in an FZ message, containing the flight path, is parsed. Characters are read until either a space or a linefeed is encountered. The arrival and departure airports (positioned at the beginning and the end, respectively, of the route text) are extracted from the route text. Any aliases for these airports are found by looking up the airport name in a hash table and following the linked-list chain to the end.

The *Route Processor* module is then called (being passed a data structure containing all the previously parsed data) to convert the route into an event list. The final event (arrival fix) is inspected to ascertain certain geographical values for that arrival fix.

The last field in an FZ message, which is optional, is the remarks field. This is free form text limited to 128 characters. The Fill_Remarks function is called to scan the field for remarks which match against any in the set of remarks contained in the NRP parameters files. Corresponding bits are set in the ac_rmks_bitflags field for any matches found.

At this point, all the data within the FZ message have been parsed, and any conversions have been made. The result is a data structure filled with values assumed to have been correctly parsed. This data structure is passed back to the calling routine and then enqueued to the

parser.relay for retrieval by the *Flight Database Processor* and enqueued to the *routedb.relay* for retrieval by the *Route Database Processor*.

Error Conditions and Handling

Any errors or perceived problems which occur during the *Parse Messages* module are logged in a diagnostic error file. Any terminal error is logged in the diagnostic error file or the parser output file. The *Parser* process then crashes and waits for the *Nodescan* process to restart the *Parser* process.

24.5.3 The Route Processor Module

Purpose

The *Route Processor* interprets the route text of each NAS message to determine the path of the flight. Based on the path, it creates a list of monitored events and waypoints. It also provides information about fixes or airports directly to the *Parser* function. The data flow diagram in Figure 24-7 shows how the *Route Processor* interacts with other areas of the *Parser*.

> **NOTE:** The route text for an individual flight is often called a *field 10* because it comes from the tenth field of the FZ or UZ message that refers to that flight. Throughout the rest of this section, the term *field 10* will be used extensively in this context.

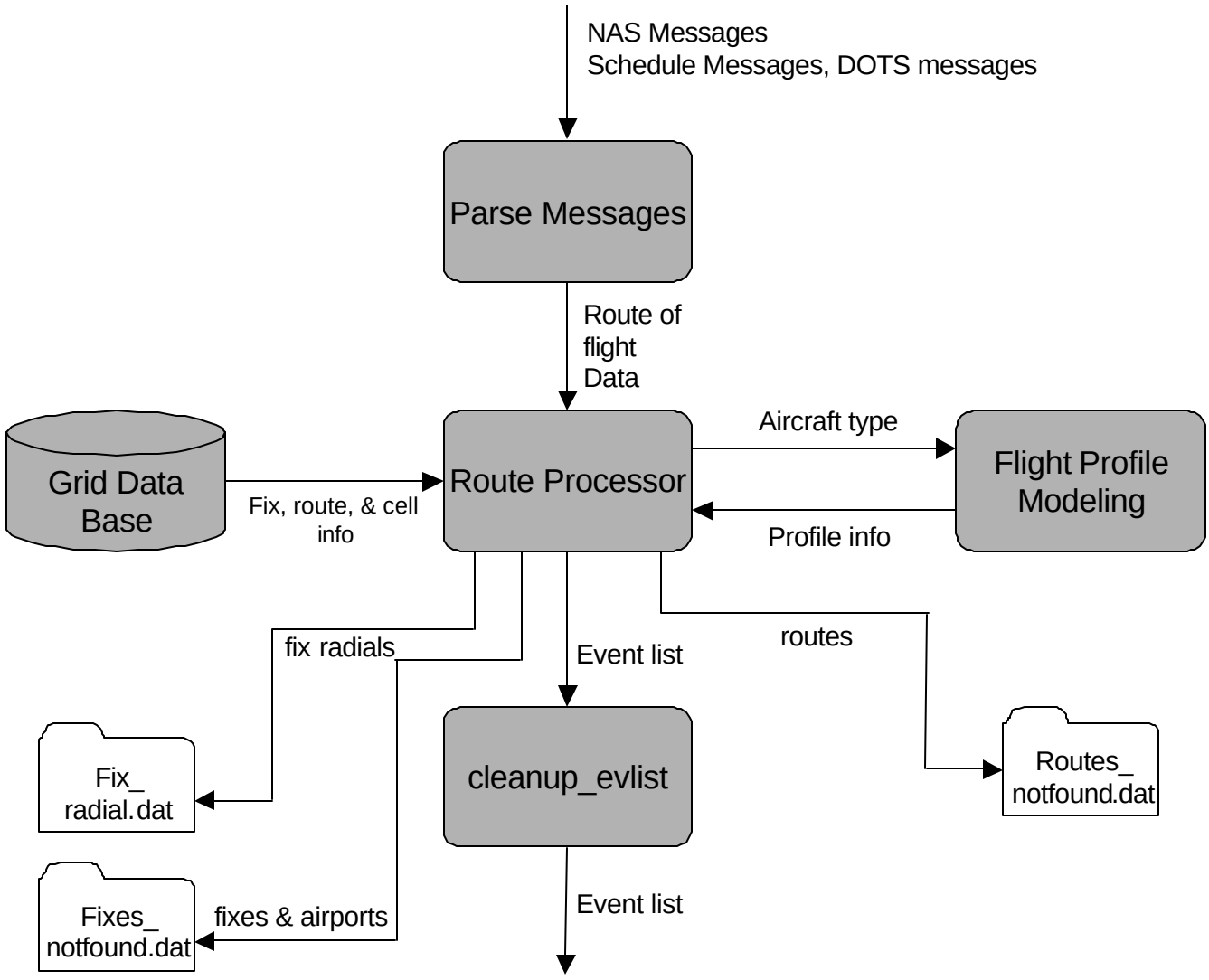


Figure 24-7. Data Flow of the Lower Parser Modules

Design Issue: Important Data Structures

All information necessary for the building of the cell list, used by the *Semantic Parser*, is saved in the **route_context** and **evidbt** record structures (see Tables 24-2 and 24-3). There are also recoverable and non-recoverable errors: their handling is explained in the individual module descriptions.

There are also two structures used to pass information about individual elements back and forth. These are the **fixid** record and the **routeid** record (see Tables 24-4 and 24-5). The first field in each, **id_type**, is used to indicate several things about the element: its type, such as named route, named fix, or latitude/longitude fix; whether it has already been successfully looked up in the database; and, for fixes, a flag to indicate whether this is either the initial or final fix in the field 10.

Table 24-2. route_context Data Structure

route_context			
Library Name: routeproc_lib		**Element Name:** griddb_interface.ins.pas	
Purpose: Stores information about each flight plan that is needed to process the route			
Data Item	**Definition**	**Unit/Format/Range**	**Var. Type**
ev	See Table 8 – 5	substructure with additional data fields	evidbt
prevalt	Previous altitude	superhi/hi/lo	element_ subtype
prevctr	Previous center	offset into mapped files	long
prevsect	Previous sector	offset into mapped files	long
prevrow	Previous row	0 – 359	short
prevcol	Previous column	0 – 839	short
inuse	Is this route context being used?	TRUE or FALSE	Boolean
generalbearing	Bearing of this flight from start point to end point	0 – 359	short
cell_head	First cell in saved linked list of pointer to gridcells	--	cel_link_ptr
cell_tail	Last cell in saved linked list of pointer to gridcells	--	cel_link_ptr
save_els	First element in saved list of events to make	--	events_to_ make_ptr
save_tail	Last element in saved list of events to make	--	events_to_ make_ptr

This flag is used because the *Syntactic Parser* makes no distinction between fixes and airports; every token found in a fix position (the odd-numbered positions in the field 10) is considered a fix. Because airports and fixes must be distinguished by the *Semantic Parser*, the *Syntactic Parser* sets a flag when sending the first and last fixes of a field 10 to the *Semantic Parser*. This flag indicates that the *Semantic Parser* should first look for this fix in the airport database, and only if the fix is not found there, in the fix database.

Table 24-3. evidbt Data Structure

evidbt	
Library Name: route_openlib	**Element Name:** fill_evist.h
Purpose: Stores information about each flight plan that is needed to process the route	

Data Item	Definition	Unit/Format/Range	Var. Type
evlist	address of event list	--	erect_arr_ptr
evpos	index to event list array	0 – MAXEVENTS	short
eltype	element type (e.f., STAR NAVAID, superhigh sector)	0 – 18	short
elkink	kind of event (e.g., arrival, departure, etc.)	0 – 2	long
elindex	external index of element	range varies by element type	short
actype	name of aircraft type	alphanumeric	string4
arrlat	latitude of destination fix	radians	float
arrlon	longitude of destination fix	radians	float
ellat	latitude of current element	radians	float
ellon	longitude of current element	radians	float
altitude	altitude at current location	100's of feet	long
elvel	velocity at current location	nautical miles/minutes	long
heading	bearing from previous event's position	0 – 359	short
deptime	departure time	0 – 2880 minutes	short
delay	delay in flight plan (as specified at certain fixes)	Minutes	long
cralt	cruising altitude	0 – 640 (100's of feet)	long
crvel	cruising velocity	nautical miles/minutes	long
monitor	Is this event monitored?	0 – 1	short
waypoint	Is this event a waypoint?	0 – 1	short
phase	flight phase	0 – 6	long
fid	flight ID	alphanumeric	string10
newfl	Is this a new flight?	TRUE or FALSE	Boolean

Table 24-4. fixid Data Structure

fixid	
Library Name: routeproc_openlib	**Element Name:** griddb_interfaceh
Purpose: Stores information about an individual fix, passes it back and forth between the syntactic and semantic parts of the route processor	

Data Item	Definition	Unit/Format/Range	Var. Type
id_type	type of fix (or airport) to look for	used bitwise to indicate type, whether it is in database	long
name	fix name as found in or created from the field 10	alphanumeric	string10
namelen	length of name	0 – 10	long
lat	fix latitude	radians	float
long	fix longitude	radians	float
magvar	magnetic variation from true North	plus or minus degrees	short
delay	delay associated with the fix in field 10	minutes	short
ap_type	non-FAR, non-FDR airport indicator (‘*’ in field 10)	0 – 1	short
offset	offset into mapped fix or airport file	range depends on file size	long
id	external index	range depends of file size	short
sub_num	numeric value for element_subtype	5 – 11	short
row	row of grid this fix is in	0 – 359	short
col	column of grid this fix is in	0 – 859	short
db_name	name of fix used in the grid database	alphanumeric	string10

Table 24-5. routeid Data Structure

routeid			
Library Name: routeproc_openlib		**Element Name:** griddb_interfaceh	
Purpose: Stores information about an individual route, passes it back and forth between the syntactic and semantic parts of the route processor			
Data Item	**Definition**	**Unit/Format/Range**	**Var. Type**
id_type	type of route to look for	bits indicate the root type and whether it is in database	long
r_start	start offset of this route—obsolete	--	long
r_end	end offset of this route—obsolete	--	long
name	route name	alphanumeric	string10
namelen	length of name	0 – 10	short
offset	offset into mapped route file	range depends on file size	long
id	external index	range depends of file size	short
sub_num	numeric value for element_subtype	12 - 16	short
circle_num	If this is a military route, number of times it circles	1 – 15	short

Input

The parts of a NAS message needed to determine the flight path are passed as input to *Route Processor* in the **nas_msg** record. This input includes

(1) Aircraft type — used for determining ascent and descent profiles

(2) Field 10 — the flight path to parse, which begins with a fix identifier and is followed by at least one unit consisting of a route identifier followed by a fix identifier. A period separates one route or fix identifier from that which follows it (i.e., *fix.route.fix.route.fix*, etc.). Sometimes one or more of the entries in the field 10 will be empty; in these cases, the fix or route is said to be *implied*. See Section 24.4.3.1 for more about *implied* routes and fixes.

(3) Cruising altitude — used for determining flight profile

(4) Cruising velocity — used for estimating times along flight

Output

The following items are produced as a result of processing routes:

(1) Status code — number of events on the event list or error code

(2) Event list — an array of records, each of which describes a waypoint or a monitored event (see Table 24-1 for details on the records).

(3) File of fix-radial routes (switch option for this in **method_file.txt**) — used to update the grid database continually

(4) File of unknown routes — used to update the grid database continually

(5) File of unknown fixes — used to update the grid database continually

Processing Overview

The *Route Processor* module's *Syntactic Parser* checks the field 10 for syntactic correctness and then calls the *Semantic Parser* to determine the flight path. When the entire path has been created, the event list is filled with waypoints and monitored elements. The event count (or negative status code if error) is returned to the *Parser* function. See Section 24.4.3.1 for details on the syntactic processing, and Section 24.4.3.2 for details on the semantic route processing. See Figure 24-8 for the data flow diagram of the route processor.

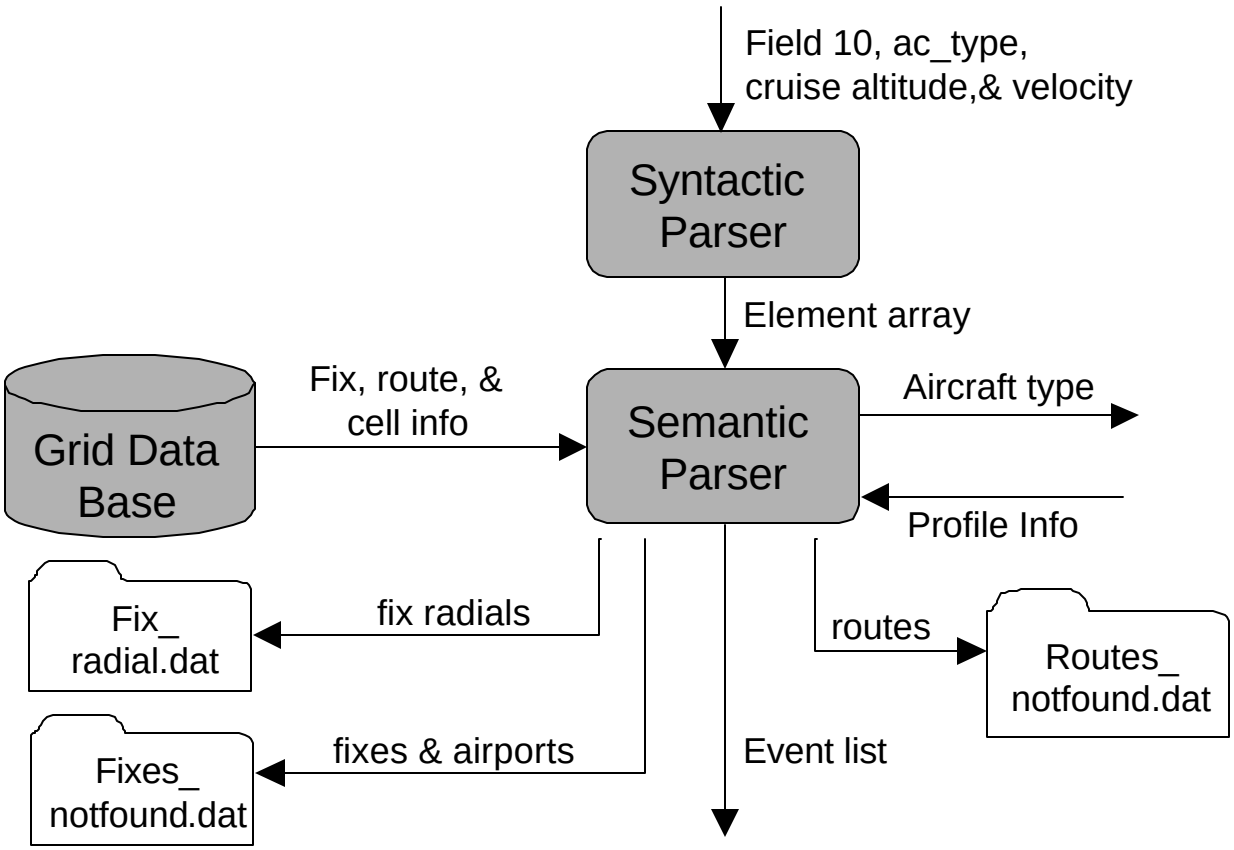


Figure 24-8. Data Flow of the Route Processor Module

24.5.3.1 The Syntactic Parser Module

Purpose

The *Syntactic Parser* checks the field 10 for legal syntax and separates the items in the field 10 by placing each of them in an array location, to be passed to the *Semantic Parser*. Based on return status, the *Syntactic Parser* calls the semantic processor. If non-recoverable errors occur, the *Syntactic Parser* passes the appropriate error code back to the *Parser*.

Input

In addition to the *Route Processor* input described in Section 24.4.3, the *Syntactic Parser* requires a pointer to the element array and an indication of the array size. This array is used to store the elements of the field 10.

Output

The *Syntactic Parser* returns a status code indicating whether or not the field 10 syntax is correct. If the syntax is correct a filled element array is also returned. If no element array could be created, an error is generated indicating that the field 10 contains a syntax error.

Processing

The *Syntactic Parser* operates as two inner state machines (one to check fix field syntax and one to check route field syntax) within a larger state machine, checking the field 10 overall for legal syntax. The route must have an odd number of fix and route tokens (including implied fixes and routes) so that it begins and ends with a fix; route and fix identifiers must match legal patterns. See Figures 24-9 through 24-11 for diagrams of field 10 syntax state machine, route syntax state machine, and fix syntax state machine operation. Except in the first or last position, an empty field is also legal.

Depending on whether it is a fix or a route field, it is known as an *implied fix* or an *implied route.* Each token is marked with its type, e.g., latitude/longitude fix, named fix, jet route, fix-radial route, empty (implied) route.

Each token is then placed in the element array which is passed to the *Semantic Parser,* through a pointer, when the syntax check is complete. If the syntax is incorrect, an error status is returned.

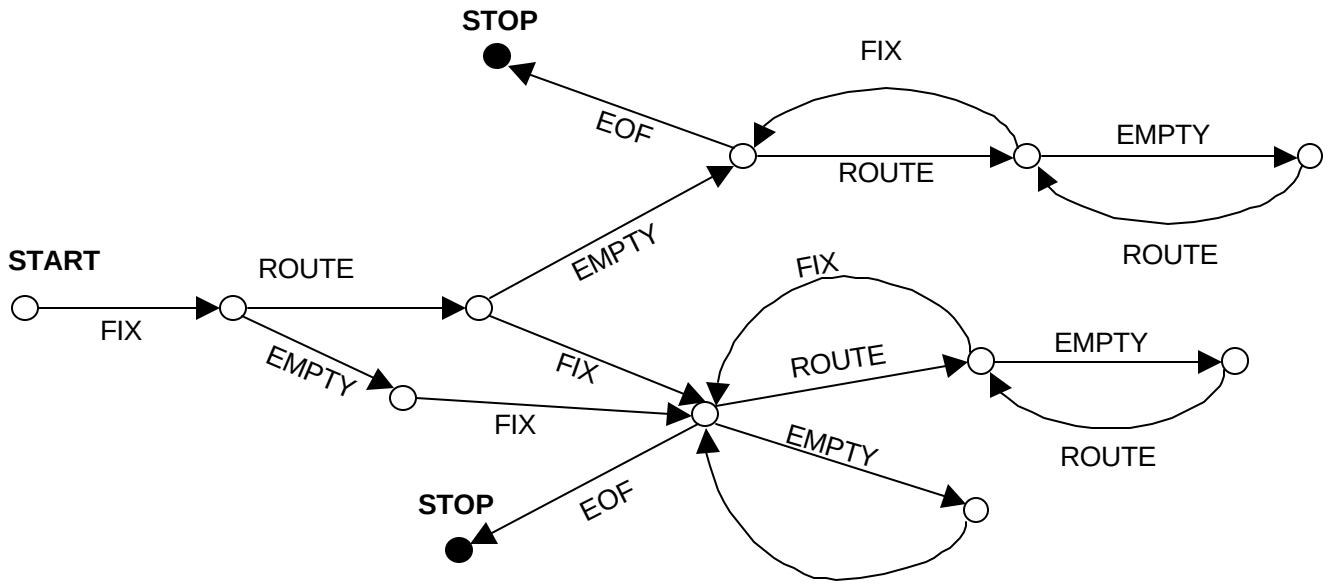


Figure 24-9. Field 10 Syntax State Machine

Error Conditions and Handling

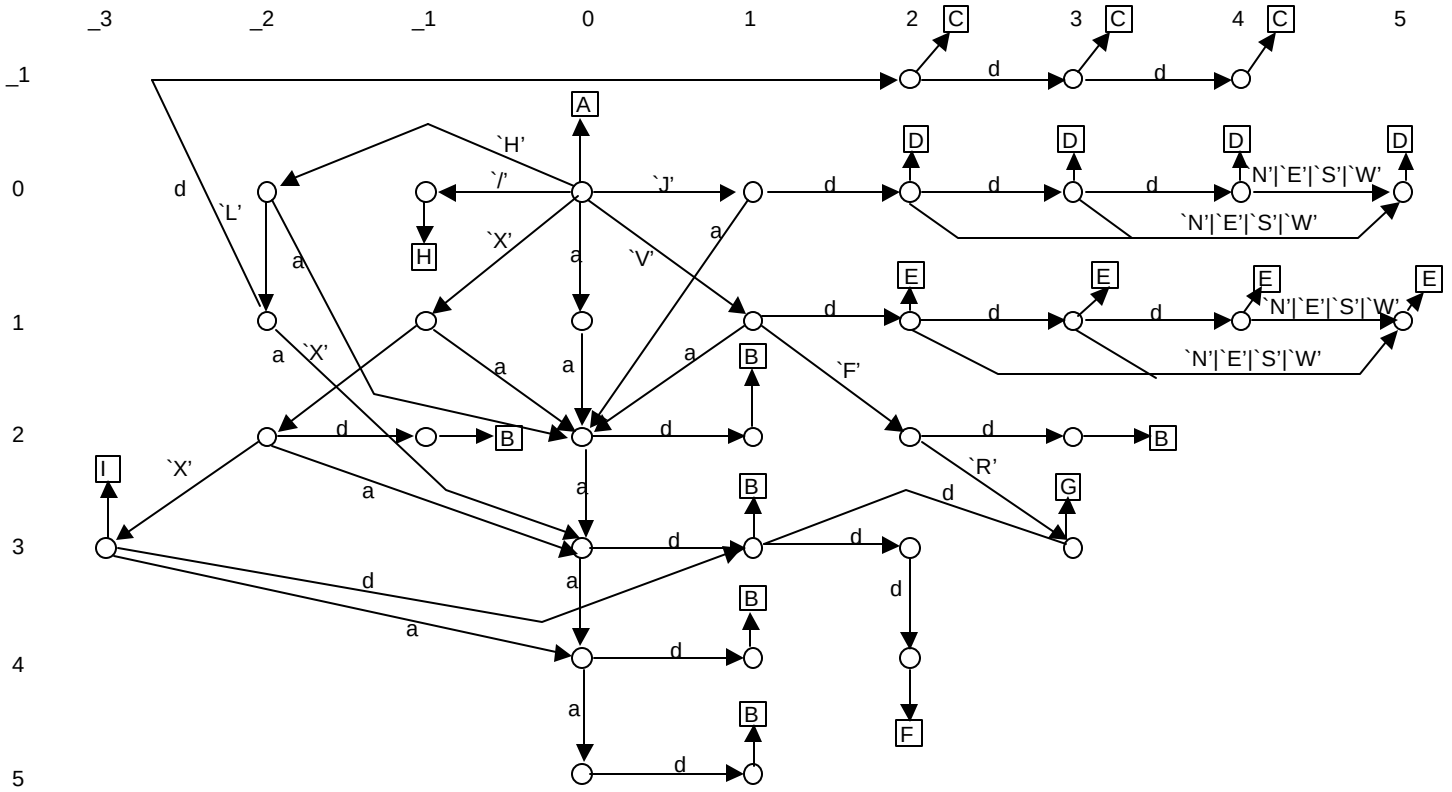
There are two types of recoverable errors associated with the *Syntactic Parser*. Descriptions of each case and the respective actions taken appear below. If these actions are unsuccessful, then the recoverable error becomes a fatal error.

(1) Unknown fix — if not in first or last position of route, the *Syntactic Parser* checks for an implied route before or after it. This error becomes fatal if one or both of the routes around the fix are implied.

(2) Unknown route — *Syntactic Parser* checks for an implied fix before or after it. This error becomes fatal if one or both of the fixes around the route are implied.

The *Syntactic Parser* has one type of fatal error, which is identified by a status code message indicating the type of error to the *Parser*. The message is explained as follows:

"Can't parse this route" — illegal syntax



KEY

The boxed letters refer to program labels for the code that completes processing for the named fix types. Each of these labels needs a period ('.') or a line-feed to be reached except for [A], which is only reached by a period.

[A]	empty_route	implied route: FIX.ROUTE.FIX. .FIX	[D]	j_route	jet route	[G] vfr	VFR – visual flight rules
[B]	sid_star	SID or STAR	[E]	v_route	victor route	[H] ti	tailoring indicator
[C]	hl_route	Canadian route	[F]	fr_route	fix radial route (fix name followed by a radial)	[I] xxx	incomplete route indicator

The column and row numbers refer to program labels. For example, column 2, row 0 of the diagram refers to label 20; column 2, row _1 refers to Label 2_1; and column _2, row 1 refers to _21.

The arrow labels `N'|`E'|`S'|`W' mean that any one of the four quoted characters will lead to the label pointed to by the arrow. That is, any one of `N', `E', `S', or `W' will lead to label 50 (column 5, row 0) or 51 (column 5, row 1)

Figure 24-10. Route Syntax State machine

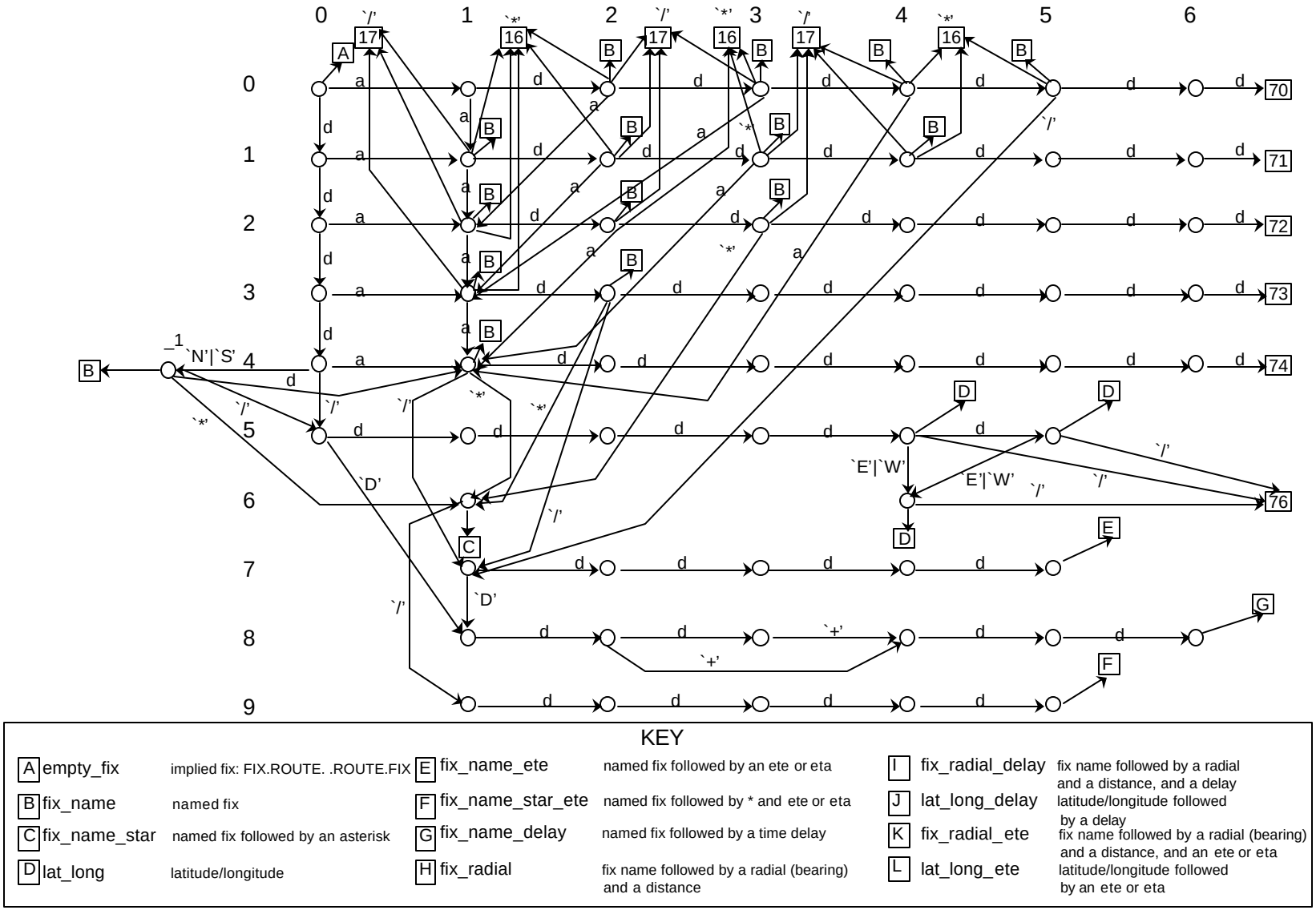
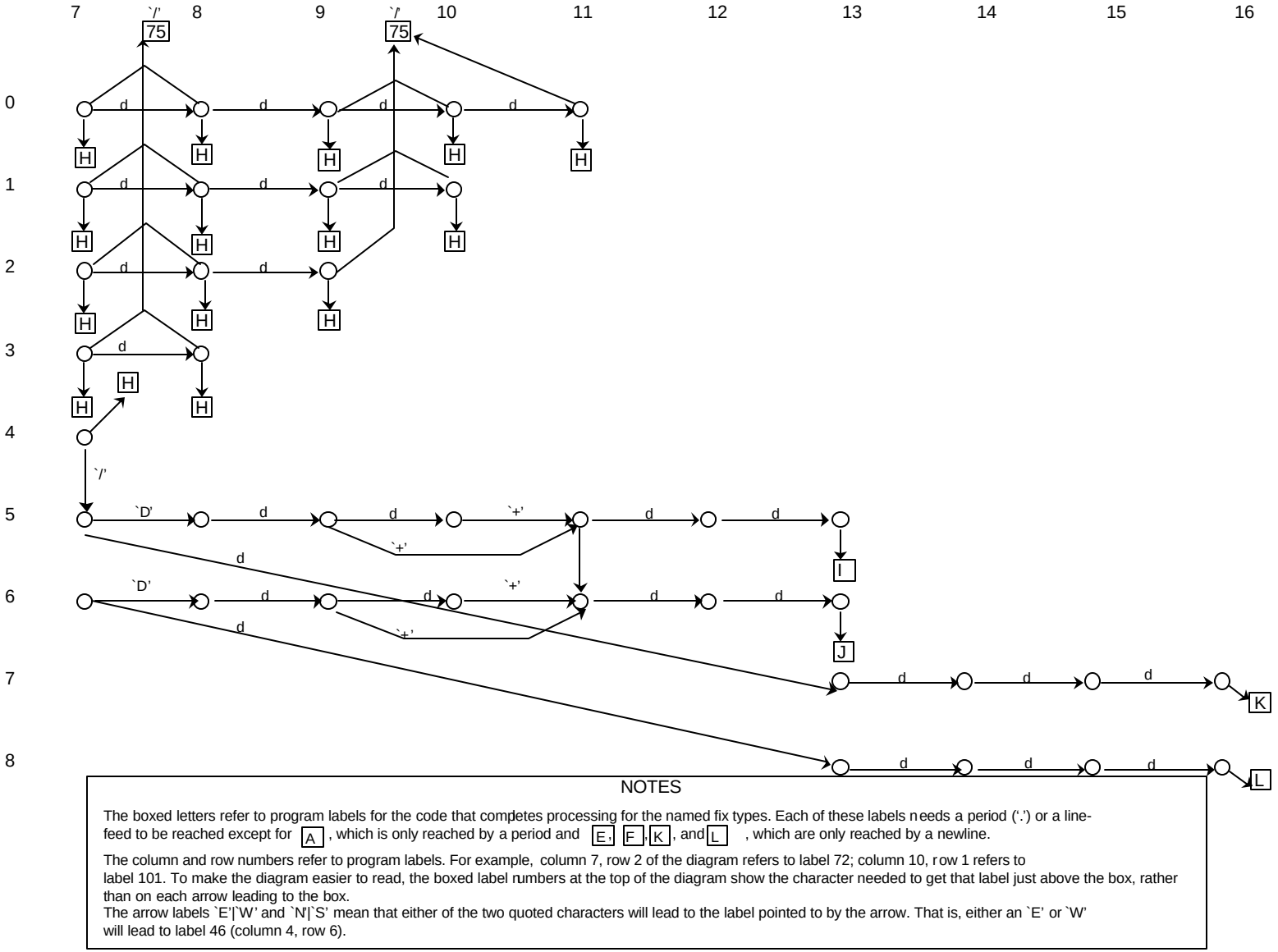


Figure 24-11. Fix Syntax State Machine



NOTES

The boxed letters refer to program labels for the code that completes processing for the named fix types. Each of these labels needs a period ('.') or a line-feed to be reached except for [A], which is only reached by a period and [E], [F], [K], and [L], which are only reached by a newline.

The column and row numbers refer to program labels. For example, column 7, row 2 of the diagram refers to label 72; column 10, row 1 refers to label 101. To make the diagram easier to read, the boxed label numbers at the top of the diagram show the character needed to get that label just above the box, rather than on each arrow leading to the box.
The arrow labels `E'|`W' and `N'|`S' mean that either of the two quoted characters will lead to the label pointed to by the arrow. That is, either an `E' or `W' will lead to label 46 (column 4, row 6).

Figure 24-11. Fix Syntax State Machine (continued)

24.5.3.2 The Semantic Parser Module

Purpose

The *Semantic Parser* uses the **grid database**, a database consisting of numerous files, to identify fixes and routes found in the field 10, to trace the path of the flight geographically, and to identify monitored elements that path enters or crosses.

Input

The *Semantic Parser* requires the element array created by the *Syntactic Parser*. It also uses the names of disk files into which routes, fixes, and fix-radial routes are to be written.

Output

The *Semantic Parser* returns a status code indicating the number of events in the event list or, if no event list could be created, the type of error that occurred. It also builds files of fixes, routes, and fix-radial routes, that were not located in the **grid database**.

Processing Overview

There are three main things accomplished by the *Semantic Parser:* the creation of **fix_route_fix** units, a **cell list**, and an **event list**. The *Semantic Parser* creates an **event list** from the element list in **two steps**. See Figure 24-12 for the data flow diagram of the *Semantic Parser*.

The *c_route* routine creates the **fix_route_fix** (**frf**) units with the help of two database lookup routines. As *c_route* traverses the element array, it calls the fix lookup routine, *db_lookup_fix,* for each fix-type token (odd-numbered positions in the field 10), and calls the route look-up routine, *db_lookup_route,* for the route-type tokens (even-numbered positions). See Sections 24.4.3.2.2 and 24.4.3.2.3 for details about the look-up routines.

If an initial fix is not found as an airport, the *c_route* assumes that the flight plan refers to a flight already in progress; i.e., not currently taking off from the ground. If the final fix is not found as an airport, the *c_route* assumes that the flight plan refers to a portion of the flight that does not include landing on the ground.

The routine that finds a pathway from fix to fix, *connect_fix_fix*, is called by the *Semantic Parser* for each **frf**-unit as it is assembled. A list of grid cells is maintained between calls to this routine. Each successful call appends cells to the list so that at the end of the field 10 there is a complete two-dimensional path for the flight, described by the list of grid cells. See Section 24.4.3.2.5 for an explanation of *connect_fix_fix*. When this list is complete, *make_event_list* is called and the **cell list** is traversed from the beginning to make the **event list**. All waypoints and any monitored elements encountered in the grid cells in the list become part of the **event list**. See Section 24.4.3.2.7 for an explanation of *make_event_list*.

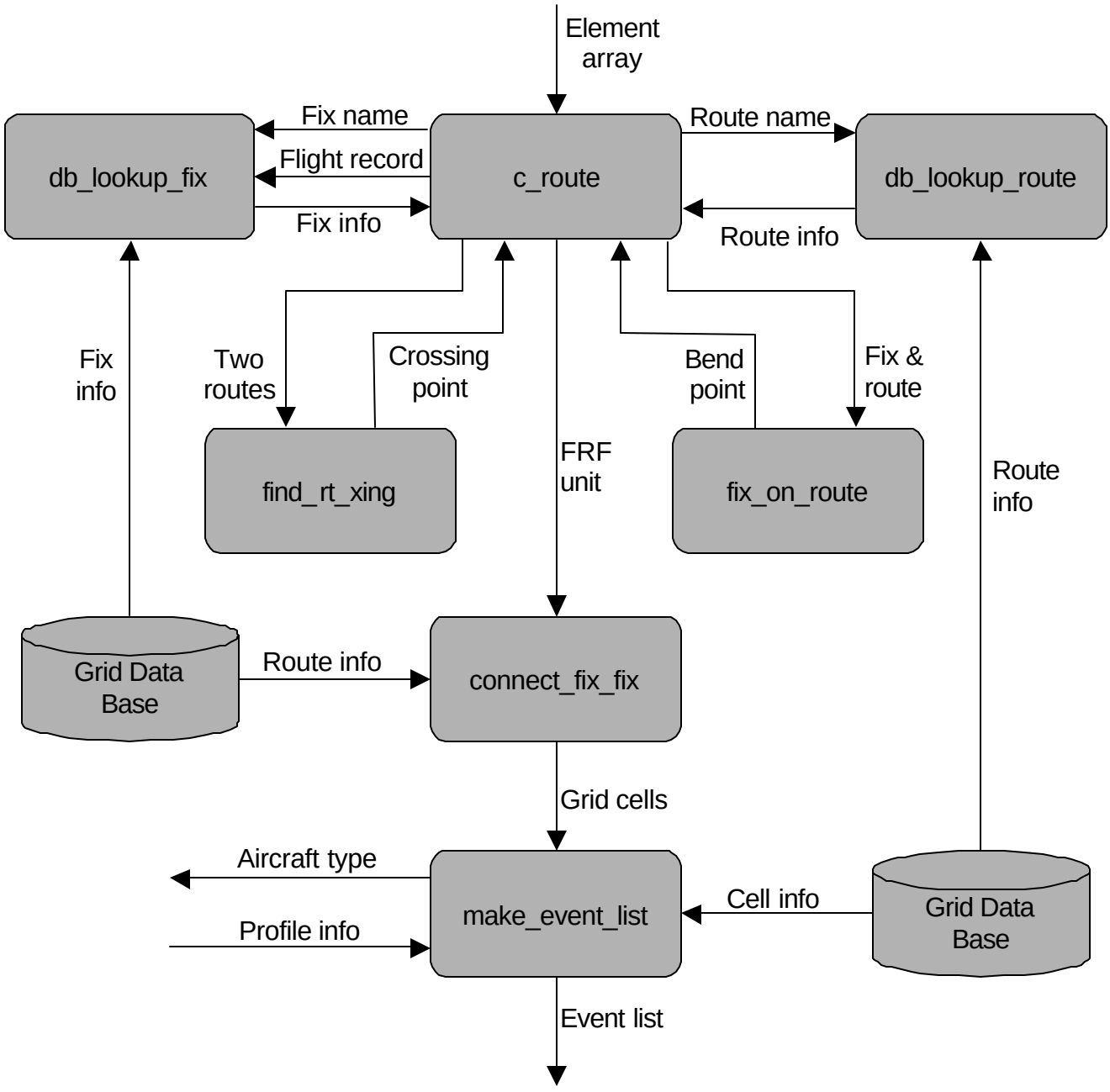


Figure 24-12. Data Flow of the Semantic Parser Module Data Flow

The *Semantic Parser* calls two other routines to allow it to create **frf**-units. When an implied or unknown fix falls between two routes, the *find_rt_xing* subroutine creates an *unnamed fix* at the latitude and longitude where the two routes intersect. See Section 24.4.3.2.4 for more information on *find_rt_xing*. The other routine needed to complete **frf**-units, *fix_on_route*, bends a route to a fix. If one of the fixes specified in an **frf**-unit is not in a grid cell the specified route passes through, an unnamed fix is created on the route in the cell closest to the specified fix. See Section 24.4.3.2.6 for more about *fix_on_route.*

24.5.3.2.1 The c_route Routine

Purpose

The *c_route* routine handles the transition from the *Syntactic Parser* to the *Semantic Parser*. It creates the **fix_route_fix** units from the element array generated by the *Syntactic Parser*. It also fills parts of the **oldfix** record used in *db_lookup_fix* (Section 24.4.3.2.2).

Input

The *c_route* routine uses a pointer to an element array which is filled with the fixes and routes that were extracted from the field 10 by the *Syntactic Parser. C_route* also receives fix and route information from look-up, crossing, and bending routines.

Output

C_route passes fix and route names, extracted from the element array, to look-up routines. It creates and passes **fix_route_fix** units to *connect_fix_fix* (Section 24.4.3.2.5). It also passes the **oldfix** record to *db_lookup_fix* to aid in the accuracy of choosing fixes correctly.

Processing

First, *c_route* receives a pointer to the element array. It then takes elements from the array, one by one, and passes their names to appropriate look-up routines. Once two fixes and a route are successfully looked up, a **fix_route_fix** unit is generated. The route may or may not be implied. This unit is passed to *connect_fix_fix* to generate a cell list.

As *c_route* works its way through the element array and calls the look-up routines, it is constantly updating the **oldfix** record to aid in the *db_lookup_up* routine's selection of fixes.

In the midst of generating these **fix_route_fix** units it may be determined that a route crossing is necessary and a call is made to *find_rt_xing* (Section 24.4.3.2.4). It is also possible that a fix may not be directly on the path of an associated route. In this case a call is made to *fix_on_route* (Section 24.4.3.2.6) to attempt a bend. If these routines are successful, a fix is passed back to *c_route* in order for it to generate two new **fix_route_fix** units, one from the first original fix to the new fix and one from the new fix to the second original fix.

Error Conditions and Handling

The *c_route* routine has two types of fatal errors, each of which is identified by a status code message indicating the type of error to the *Semantic Parser*. The messages are explained below:

(1) “Can't open route” — syntax was legal, but either so many fixes or routes were unknown that the route got shortened to nothing, or the initial fix was unknown.

(2) “Encountered route or fix type not supported” — syntax was legal, but the type of element was unknown or incorrect for that location in the flight path.

24.5.3.2.2 The db_lookup_fix Routine

Purpose

The *db_lookup_fix* routine determines whether the fix is known to the **grid database** (Figure 24-13), if it has duplicates, and if it is within the logical range of the flight path. If so, it fills in the fields of the **fixid** record (Table 24-4) such as latitude/longitude, row, and column on grid cell, index, magnetic variation, etc. See Figure 24-14 for an illustration of the logical flow of *db_lookup_fix*.

Input

The *db_lookup_fix* routine expects a pointer to a **fixid**-type record with at least the **id_type** field and either the **name** or the **latitude** and **longitude** fields filled in. It also expects a pointer to an **oldfix**-type record (see Table 24-6) that contains information about the flight.

Output

The routine passes back values of **true** or **false** based on whether the fix is found. If a fix is found, all fields of **fixid** type record are filled in, including **id_type** bits to indicate the fix's type and its "found" status (to prevent duplicated effort in potential future look-ups).

Processing

First, *db_lookup_fix* checks the **id_type** field of the **fixid** record to see whether the bit that indicates that this fix has already been looked up successfully has been set. If so, the routine exits, returning a value of **true**.

If the **id_type** indicates that this is a latitude/longitude fix, the routine computes the grid cell of this latitude/longitude and looks for a fix with the identical latitude/longitude in that cell. If one is found, the *fix_verify* routine is called to verify the fix is at a valid location, with respect to the bearing and distance of the flight path. If it is, the fields in the **fixid** record are filled and the routine exits, returning a value of **true**. If the fix is not confirmed, and determined to be out of range, the routine exits, returning a value of **false**, and the fix is eventually tossed.

If a fix with the identical latitude/longitude is not found, the fix lookup routine creates an **unnamed** fix, calls *fix_verify* to confirm the **unnamed** fix's location, in respect to the flight path. If verified, the routine fills in the **fixid** fields and exits, returning a value of **true**. If the fix is not verified, and determined to be out of range, the routine exits, returning a value of **false**, and the fix is eventually tossed.

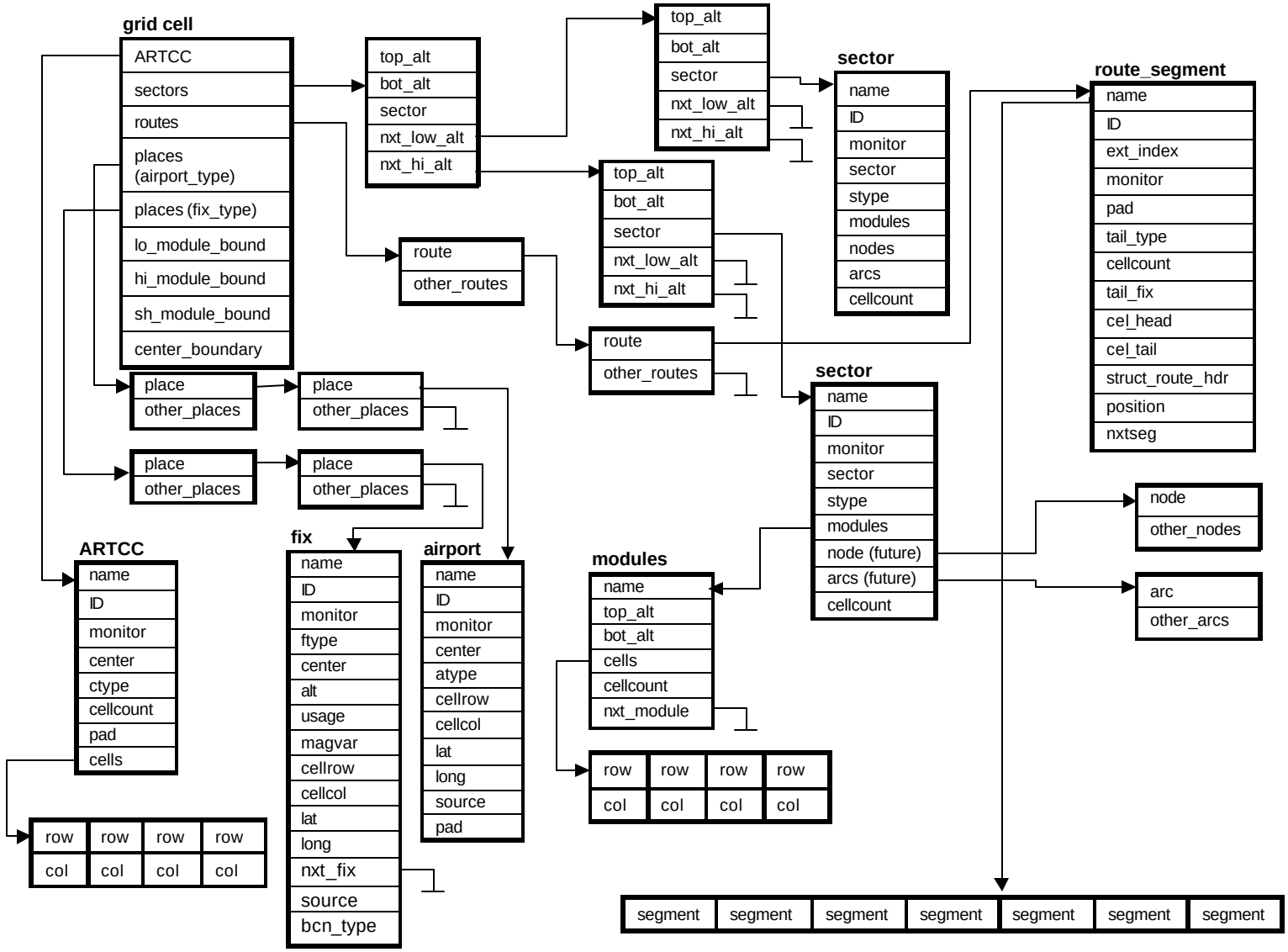


Figure 24-13. Grid Database

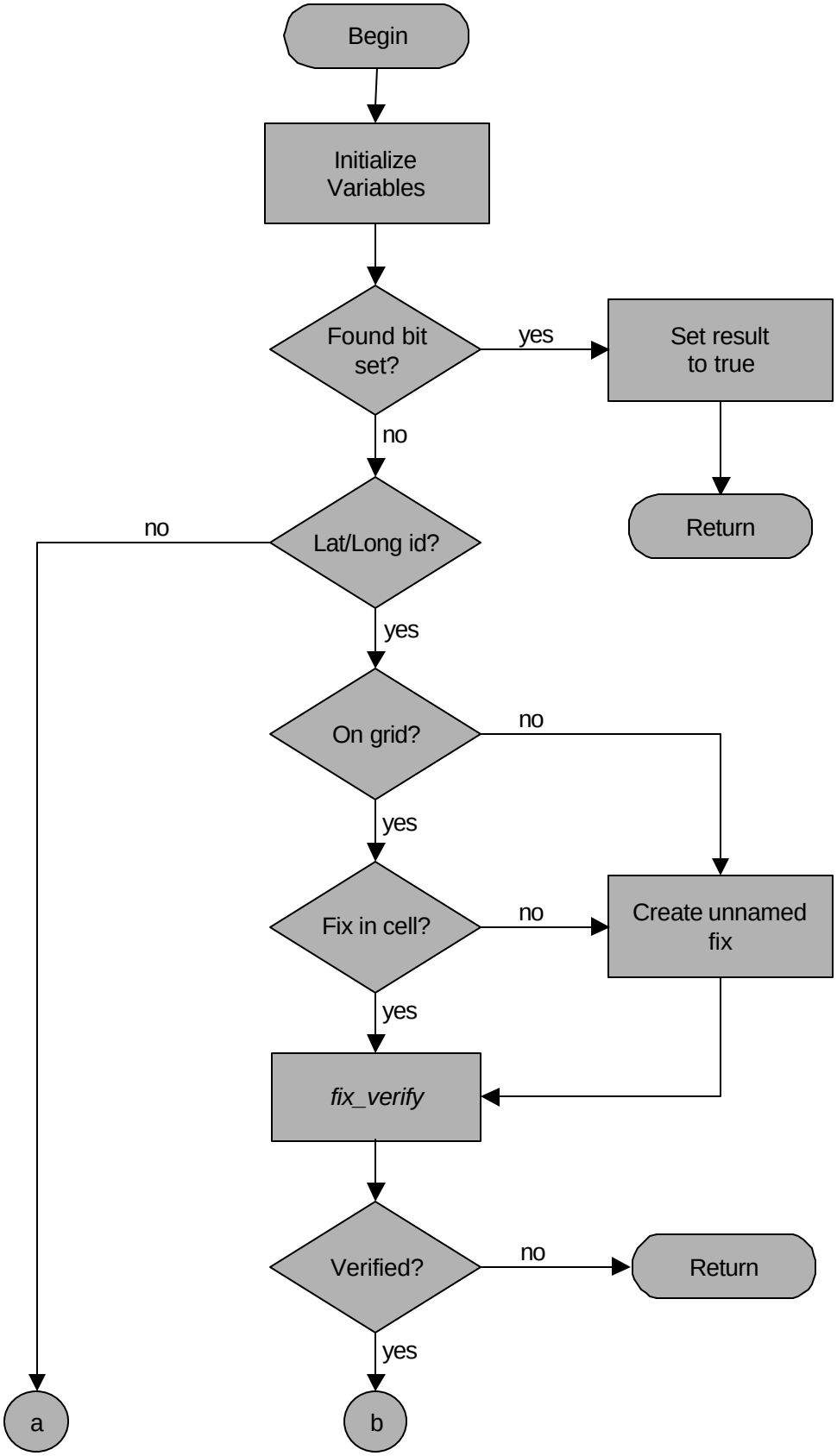


Figure 24-14. Sequential Logic for the db_lookup_fix Routine

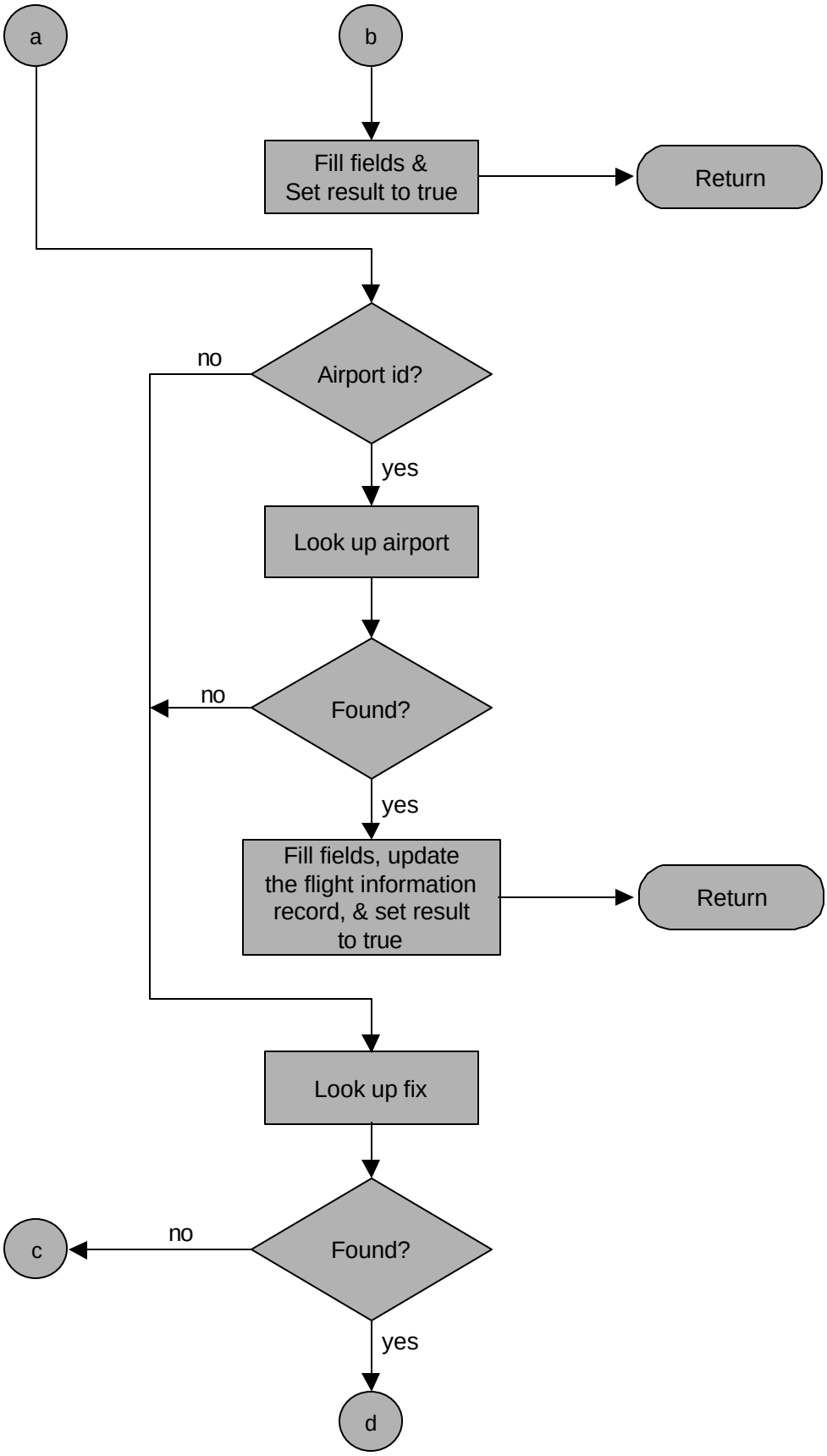


Figure 24-14. Sequential Logic for the db_lookup_fix Routine (continued)

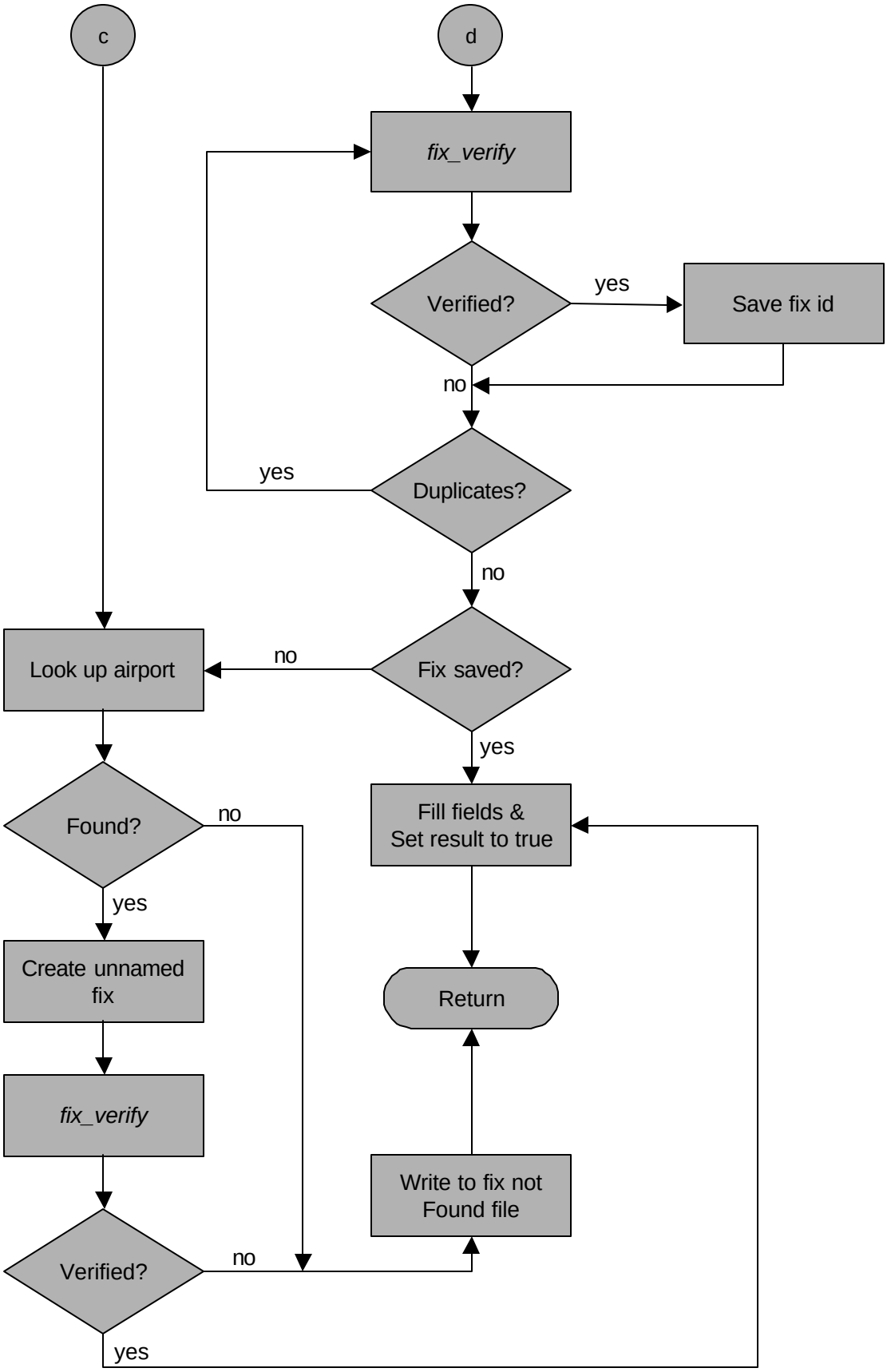


Figure 24-14. Sequential Logic for the db_lookup_fix Routine (continued)

Table 24-6. oldfix Data Structure

oldfix			
Library Name: routeproc		**Element Name:** gridbd_interfaceh	
Purpose: Stores information about a flight needed to determine the correct fix			
Data Item	**Definition**	**Unit/Format/Range**	**Var. Type**
lat	Latitude of last accepted fix	radians	float
long	longitude of last accepted fix	radians	float
routeflag	type of element that precedes the presnt fix	1 – 16	long
rte_name	name of the route that precedes the present fix	alphanumeric	string10
routeflag2	type of element that follows the present fix	1 – 16	long
rte_name2	name of the route that	alphanumeric	string10
f_direct	bearing of the flight	0 – 359	long
a_dist	actual proposed distance	nautical miles	float
c_dist	cumulative distance of the	nautical miles	float
dep-name	departure airport latitude	alphanumeric	string10
dep-lat	departure airport latitude	radians	float
dep-long	departure longitude	radians	float
arr_name	arrival airport name	alphanumeric	string10
arr_lat	arrival airport latitude	radians	float
arr_long	arrival airport longitude	radians	float

If the **id_type** indicates that this is an airport, the name is hashed into the airport mapped files. If the airport name is not found, it is hashed into the fix mapped files. If the name is found either as an airport or a fix, the fields of the **fixid** record are filled, and the routine exits returning a value of **true**. If the **id_type** initially indicates an airport, *fix_verify* is not called.

If the **id_type** indicates that this is a named fix, the name is hashed into the fix mapped files. If the name is found, *fix_verify* is called to confirm the fix's location, with respect to the bearing and distance of the flight. Whether it was confirmed in *fix_verify* or not, the fix is then checked for duplicates. If a fix was confirmed in respect to the flight path, it is saved, and any duplicate must essentially be in a better location to be selected over the previously saved fix. When no duplicates remain, the confirmed fix (with the best location regarding the flight) will be saved to use in the flight plan. The **fixid** fields are filled; the routine exits and returns a value of **true**.

If no fix was found or confirmed, the named fix is hashed into the airport mapped files. If an airport exists with this name, an **unnamed** fix is created, and a call is made to *fix_verify* to confirm the **unnamed** fix's location regarding the flight path. If confirmed, the **fixid** fields are filled; the routine exits, and it returns a value of **true**.

The *fix_verify* routine is used to confirm a fix location with respect to the bearing and distance of the flight. This routine increases the accuracy of fixes being accepted. In order to be accepted, not only must the fix exist in the **grid database**, it also must pass distance and bearing checks. This allows flight paths to be more accurate, by choosing between good and bad fixes. The *fix_verify* routine uses the **oldfix** record to obtain the flight information. It compares the current fix to a previously accepted fix in the flight path. If the distance and bearing of these fixes are within the limits of the flight's distance and bearing, the fix will be accepted. Any other duplicate fixes that follow must also fall within these limits, but must be closer along the flight path than a previously accepted duplicate. All duplicate fixes will be checked before any one can be used in a **frf**-unit.

Error Conditions and Handling

If no fix is found, the *db_lookup_fix* routine returns a value of **false**. It also writes the fix name into the **fixes-not-found** file, marking it with an asterisk if it was looked for as an airport and an exclamation point if it was found but not verified. The departure and arrival airports are also written to the file to help determine where the fix was trying to be used. This file aids in keeping the **grid database** up to date with the flight plans being received.

24.5.3.2.3 The db_lookup_route Routine

Purpose

The *db_lookup_route* routine determines whether a named route or a fix-radial route is known to the **grid database**. The route lookup routine fills in unfilled fields in the **routeid** record (Table 24-5), such as the internal index and subtype number fields.

Input

The *db_lookup_route* routine uses a pointer to a **routeid** type record with at least the **id_type** field and the name filled in.

Output

Db_lookup_route passes back values of either **true** or **false** based on whether the route has been found. If a route is found, all fields of **routeid** type record are filled in, including an **id_type** bit to indicate that this route has been successfully looked up (to prevent duplicated effort in potential future lookups).

Processing

First, *db_lookup_route* checks the **id_type** field of the **routeid** record to see whether the "route was found" bit is set. If so, the routine exits, returning a value of **true**.

Otherwise, the route look-up routine hashes the name into the structured routes mapped file. If the route name is found, the fields of the **routeid** record are filled, including the bit in **id_type** that indicates a successful look-up; the routine exits, and returns a value of **true**.

Error Conditions and Handling

If the route is not found, the routine returns a value of **false**. If it is not a fix-radial route, its name is entered into the **routes-not-found** file. A fix-radial route that is not found in the database is created and entered into a text file, **fix-radial** file, to be added to the database later.

24.5.3.2.4 The find_rt_xing Routine

Purpose

The *find_rt_xing* routine is used to find the point at which two routes intersect. It is called when the fix between two known routes is either implied or unknown. It creates an unnamed fix at the latitude/longitude where the two routes intersect.

Input

Find_rt_xing receives a pointer to the **fixid** record (to fill if the route intersection is found) and receives pointers to the **routeid** records. Also passed are two arguments which are now obsolete: a pointer to a fix around which to search for the route intersection, and an integer indicating the radius length, measured in numbers of grid cells.

Output

Find_rt_xing returns a status code indicating whether an intersection was found. If one was found, the fields of the **fixid** record are filled in.

Processing

Starting at the last known grid cell (saved in the **route_context** fields **prevrow** and **prevcol**), *find_rt_xing* looks for the first route, searching first on the general bearing that the flight path is taking. If the first route is found, the routine searches along the first route for the second route. *Find_rt_xing* starts searching at the point closest to the last known location, looking first in the direction of the general bearing of the flight path, until it finds the second route or reaches the end of the first route. If the second route is not found anywhere along the first route in the

direction of general bearing, the search starts again at the initial point, going in the opposite direction toward the other end of the route. If the second route is found, the routine fills in the fields of the **fixid** record with the latitude and longitude of the grid cell in which the intersection is located.

The intersection of two fix-radial-routes can be calculated to get this latitude and longitude even if the two routes do not exist as entries in the database. Otherwise, the intersection of any two routes in the database can be found (if they do intersect). Intersections between routes in the database and fix-radial-routes not in the database are not calculated. If an intersection is successfully found, the unnamed fix created becomes one end of an **frf**-unit.

Error Conditions and Handling

The *find_rt_xing* routine has one type of fatal error, which is identified by a status code message indicating the type of error to the *Semantic Parser*. The message is explained as follows:

"No route intersection found" — routes specified do not intersect.

24.5.3.2.5 The connect_fix_fix Routine

Purpose

The *connect_fix_fix* routine calls a number of routines in order to create a linked list of grid cells that trace the path of the flight. The *connect_fix_fix* routine is the **first** of two passes required to create the **event list**. This pass creates the **cell list** from the **frf**-units. See Figure 24-15 for an illustration of the logical flow used in *connect_fix_fix*.

Input

There are pointers to two **fixid** records for the two fixes to be connected, a pointer to the **routeid** record of the route along which to connect them (or **nil**, if there is no route), pointer to the **event list** array, and an index into the **event list** array are the input arguments to the *connect_fix_fix* routine. Much information needed for successive calls to the *connect_fix_fix* subroutines is saved in the **route_context** and **evidbt** records (see Tables 24-2 and 24-3).

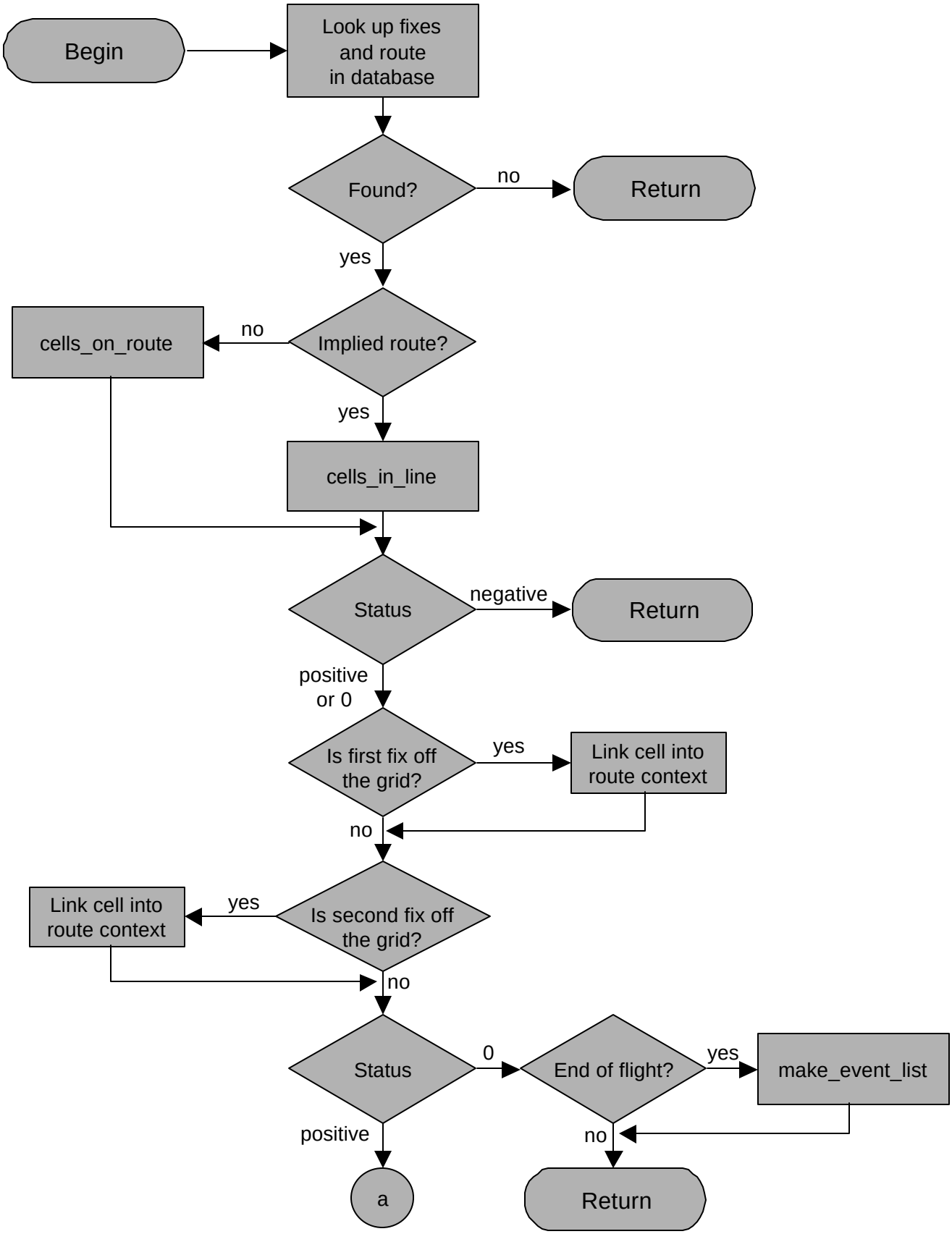


Figure 24-15. Sequential Logic for the connect_fix_fix Routine

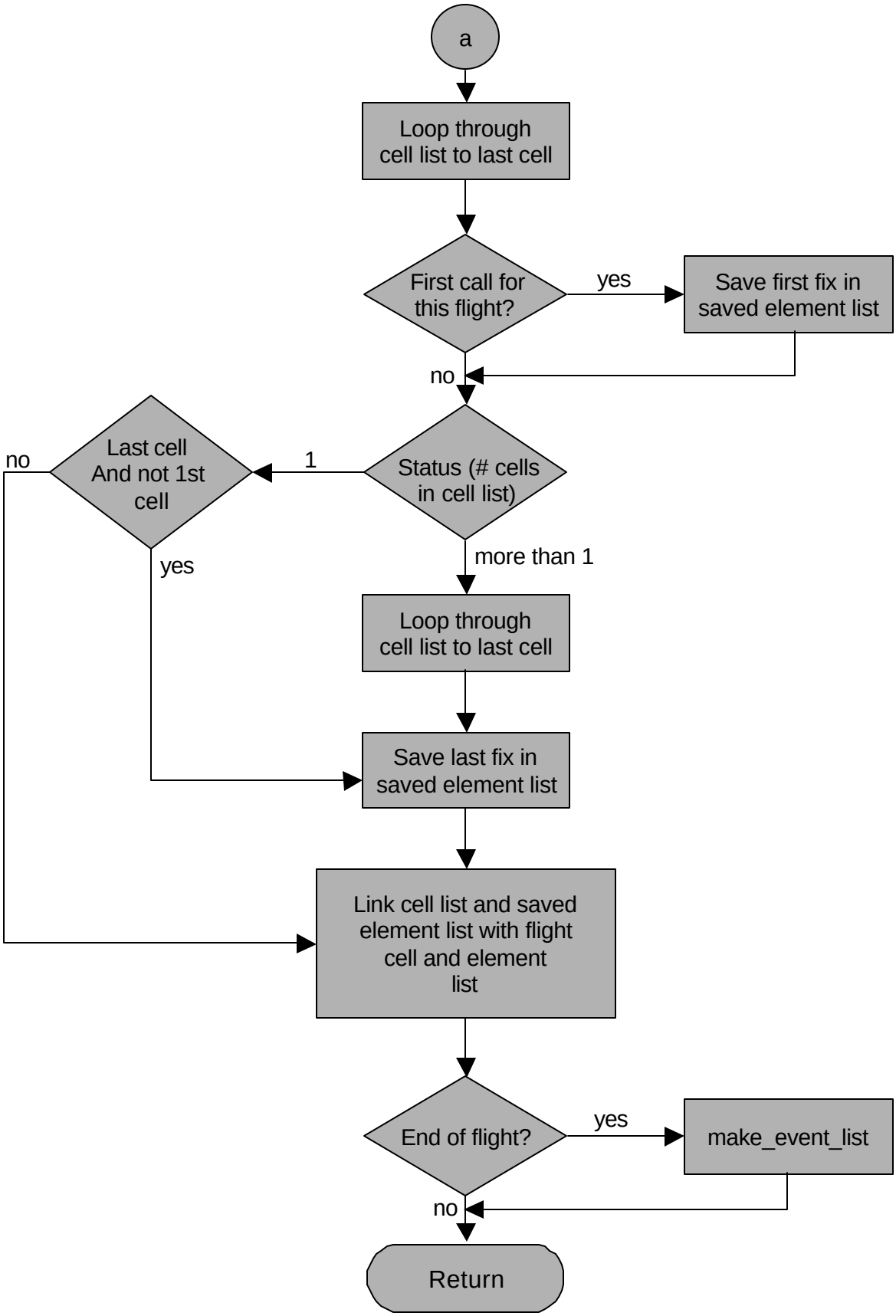


Figure 24-15. Sequential Logic for the connect_fix_fix Routine (continued)

Output

Connect_fix_fix returns a status code. A positive number indicates the number of grid cells connected between the two fixes and route (if specified) passed into the routine; a negative number indicates the type of error that occurred. It also passes a linked list of grid cells to *make_event_list*.

Processing

The *connect_fix_fix* routine first checks an **frf**-unit to see if both fixes and the route (if specified) are known to the database by calling the fix and route look-up routines (Sections 24.4.3.2.2 and 24.4.3.2.3).

Next, it creates a linked list of grid cells between the two fixes that need to be connected. If no route is specified, the linked list is created by calling the *cells_in_line* routine (Section 24.4.3.2.5.1). The cell coordinates are saved in linked **cel_link_lst** records (see Table 24-7).

Table 24-7. cel_link_lst Data Structure

cel_link_lst			
Library Name: db_rteproc_openlib		**Element Name:** griddb.decl.typesh	
Purpose: Used to store linked lists of gridcells			
Data Item	**Definition**	**Unit/Format/Range**	**Var. Type**
cel	substructure with additional data fields	-	cel_lst
waypoint	Is this a waypoint cell along a flight path?	0 – 1	short
pad	just a pad for alignment	-	short
fixoffset	offset into map file for fix at the waypoint	range depends on file size	long
link	pointer to next record of this type	-	cel_link_ptr

If a route has been specified, *connect_fix_fix* determines the **cell list** by calling the *cells_on_route* routine (Section 24.4.3.2.5.2; see Table 24-8).

At the end of each **frf**-unit, *connect_fix_fix* sets the corresponding cell in the **waypoint** field of **cel_link_lst**, and if the **tail_type** of the **route_segment** is a fix, the **tail_fix** is copied to the cell list's **fixoffset** field, which is used later to add waypoint fixes to the event list. If the second cell has not been reached, the next **segment** in the **structured_route** is used to find the **route_segment** that continues the route, and its cells are added to the **cel_link_lst**. This continues until the goal cell is reached.

Table 24-8. cel_ lst Data Structure

cel_ lst			
Library Name: db_rteproc_openlib		**Element Name:** griddb.declarationsh	
Purpose: Used to represent a single tridcell's coordinates			
Data Item	**Definition**	**Unit/Format/Range**	**Var. Type**
row	row coordination in grid	0 – 359	short
col	column coordinate in grid	0 - 839	short

Using either the *cells_in_line* method or the *cells_on_route* method, *connect_fix_fix* stores the distance along the **cell list** in a global variable as it goes along. In lists created by *cells_in_line*, this is a single distance from start to finish. In lists created by *cells_on_route*, the distances are calculated from start to finish of each part of each route segment that is added to the list; they are added together to get the total distance along that part of the route.

Next, *connect_fix_fix* loops through the cells on the **cell list** (if any), saving the fix tokens associated with the beginning and the end of the **frf**-unit in a linked list of **events_to_make** records (Table 24-9). *Connect_fix_fix* "hooks in" both ends of the **cell list** and both ends of the **events_to_make** list to the **route_context** record.

Table 24-9. events_to_make Data Structure

events_to_make			
Library Name: routeproc_openlib		**Element Name:** griddb.interfaceh	
Purpose: Used to store lists of airports and fixes for which to make events			
Data Item	**Definition**	**Unit/Format/Range**	**Var. Type**
element	pointer to fix or airport for which to make an event	-	fixid_ptr
link	pointer to next record of this type	-	events_to_make_ptr

If *connect_fix_fix* is working on the last **frf**-unit for the field 10 in question (indicated by the **id_type** code in the **fixid** record of the second fix), then it calls the *make_event_list* routine (Section 24.4.3.2.7) to create the **event list**.

24.5.3.2.5.1 The cells_in_line Routine

If the distance between the two fixes is less than approximately 180 nautical miles, the *cells_in_line* subroutine selects the cells by calculating the slope between the two fixes to get a series of row and column coordinates, which are saved in linked **cel_link_lst** records (see Table 24-7). If the distance exceeds 180 n.m., the *greatcircle_connect* subroutine determines the row and column coordinates by calculating the great circle between the two fixes. *Cells_in_line* also saves these row and column coordinates in the **cel_link_lst**.

24.5.3.2.5.2 The cells_on_route Routine

The *cells_on_route* subroutine starts in the grid cell of the first fix, getting the reference to the route segment that passes through that cell. It uses the **position** field of the **route_segment** record to find the point along the structured route where this route segment falls. Starting with the row and column of the first fix, *cells_on_route* puts cells in the **cel_link_lst** by tracing along the **route_segment** record's list of **cels** (see Table 24-7) in the direction from the cell which contains the first fix to the cell which contains the second fix.

Error Conditions and Handling

There is one type of recoverable error associated with the *connect_fix_fix*. The following describes the error and its respective action taken.

> Fix not on route — if a fix is not exactly on the specified route, then the *fix_on_route* routine is called. The route is bent and the closest point on the route to the fix is found. This point is then used as the other fix in the frf-unit, connecting to the original fix with an implied route. After this frf-unit is successfully connected, another frf-unit is made to connect the newly made fix with the other fix in the original frf-unit along the route.

If the action is unsuccessful, the recoverable error could become a fatal error.

The *connect_fix_fix* routine has two types of fatal errors, each of which is identified by a status code message indicating the type of error to the *Semantic Parser*. The messages are explained below:

(1) "Memory full" — memory was too full to allow the route to be processed

(2) "Error after route bent" — grid cells could not be connected after a route has been bent to a fix

24.5.3.2.6 The fix_on_route Routine

Purpose

When the fix specified in the field 10 is not exactly on the specified route, *fix_on_route* attaches the fix to the route by bending the route within a specified radius limit to meet the fix. See Figure 24-16 for an illustration of the logical flow used in *fix_on_route.*

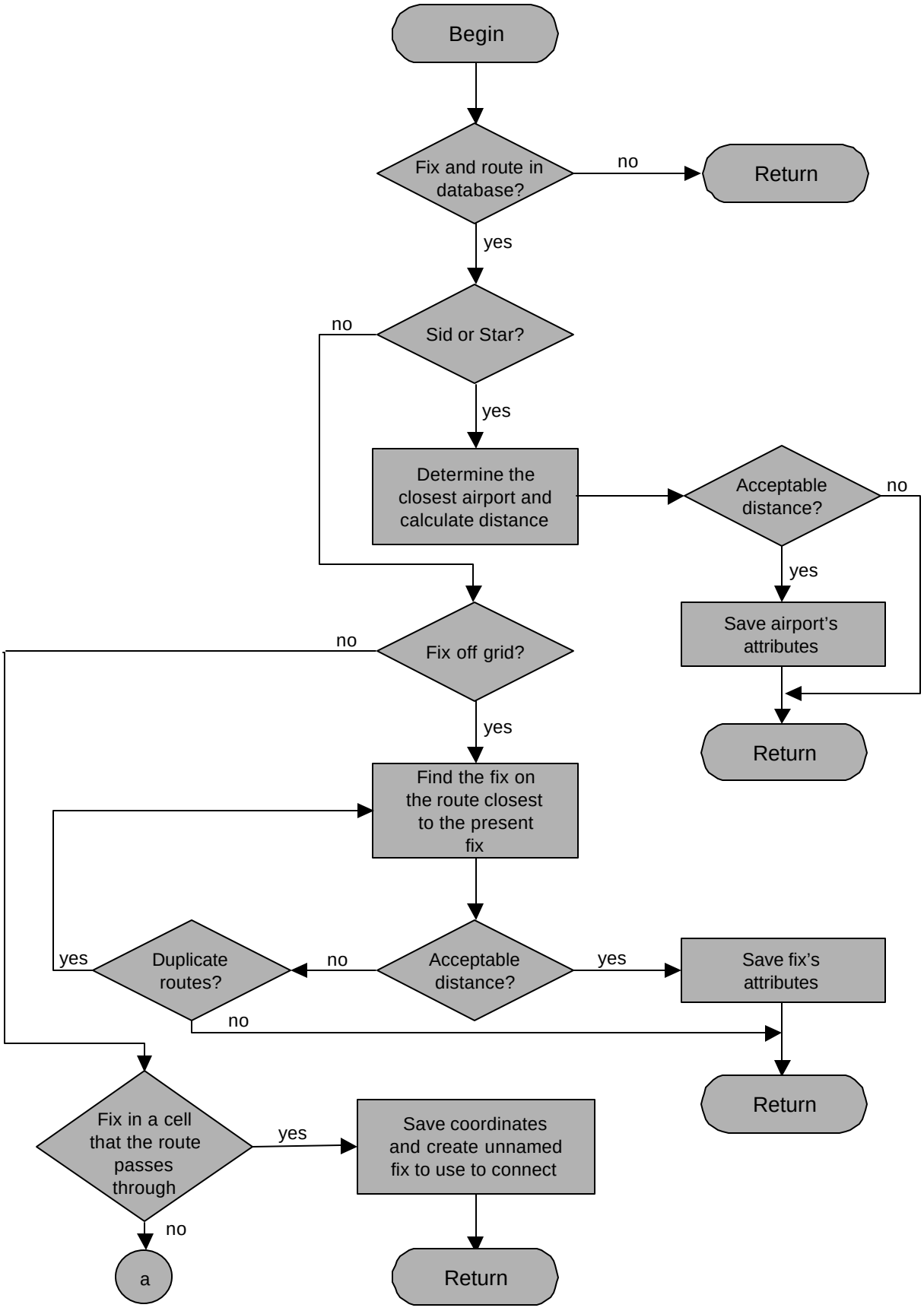


Figure 24-16. Sequential Logic for the fix_on_route Routine

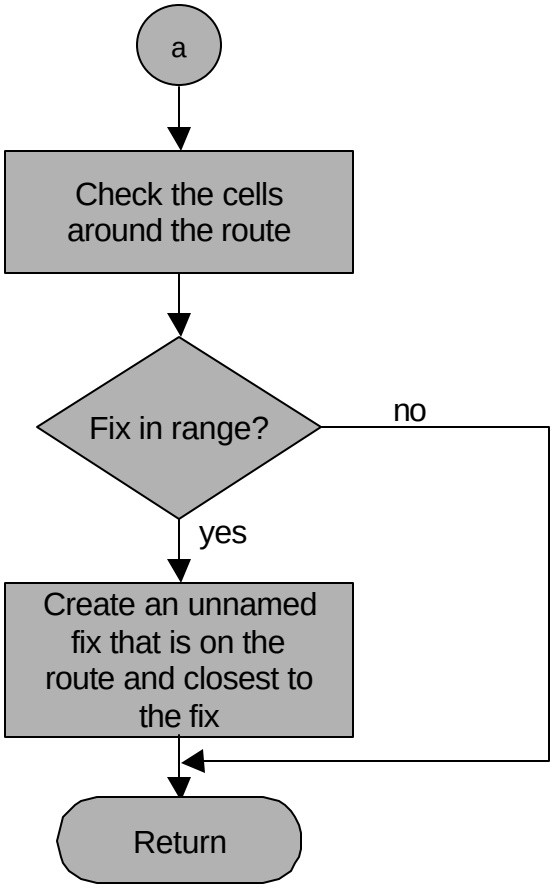


Figure 24-16. Sequential Logic for the fix_on_route Routine (continued)

Input

Input to *fix_on_route* includes a pointer to the **fixid** record of the fix to which the route must bend, a pointer to the **routeid** record of the route to bend, an integer indicating the radius length, measured in numbers of grid cells, to search for the route, and a pointer to the **fixid** record to fill if the route is found.

Output

Fix_on_route returns a value of either **true** or **false** depending on whether the route is found within the specified radius. If the route is found, the fields of the second **fixid** record are filled.

Processing

If the route is a SID or a STAR, the grid cell at the end of the route that is closest to the related airport is located and used as the new fix. If the fix to which the route needs to be bent is off the grid, the grid cell on the route that is closest to the fix (on the edge of the grid) is found and used as the new fix. Otherwise, *fix_on_route* searches around the fix within the grid cell radius limit to find the closest grid cell that the route passes through. If one is found within the radius limit, the fields of the **fixid** record are filled in with the latitude and longitude of the grid cell found.

Next, the *c_route* routine creates two **frf**-units, one with the specified fix, an implied route, and the newly created unnamed fix, and another one consisting of the unnamed fix, the specified route, and the other specified fix from the original **frf**-unit. This bending can be done to the first fix, the second fix, or, if necessary, to both fixes in an **frf**-unit.

Error Conditions and Handling

The *fix_on_route* routine has one type of error, which is identified by a status code message indicating the type of error to the *Semantic Parser*. The text explains the message:

> "Can't bend route to fix" — fix and route specified adjacent to each other in flight plan are not within a predetermined distance

The routine returns a value of **false** and the status code, if the route can't be bent to the fix.

24.5.3.2.7 The make_event_list Routine

Purpose

The *make_event_list* routine creates the list of waypoints and monitored events once the list of grid cells is complete. The *make_event_list* routine is the **second pass** of the flight elements needed to create the **event list**. See Figure 24-17 for an illustration of the logical flow used in *make_event_list*.

Input

A pointer to the event list array is passed from *connect_fix_fix*. Information needed for the *make_event_list* routine is saved in the **route_context** record. Some additional input is kept in files. The **fix priority** data file is read into a global array during initialization. This array is used to determine which type of fix to use as a waypoint in the event list in the case of a grid cell with multiple unmonitored fixes. The **activetypes** data file is read into a set in initialization. It is used to save processing time by indicating whether a whole element type is unmonitored, eliminating the need to search for events of that type.

Output

Make_event_list returns a status code. A positive number indicates the number of events on the event list; a negative number indicates the type of error that occurred. The event list is filled with records describing all the waypoints and monitored events in the flight.

Processing

Make_event_list first calls *assign_a_profile* (Section 24.4.4.2) to initialize the aircraft profile information for this flight. Then, starting at the beginning of the **cel_link_lst** and the beginning of the **events_to_make** list, *make_event_list* makes events for any fixes that have grid cell coordinates matching those of the first cell in the **cell list**. Distances between waypoints get added to the cumulative distance traveled along the **cell list** and put into the **now.dist** field of the **a_flight** record (See Table 24-10).

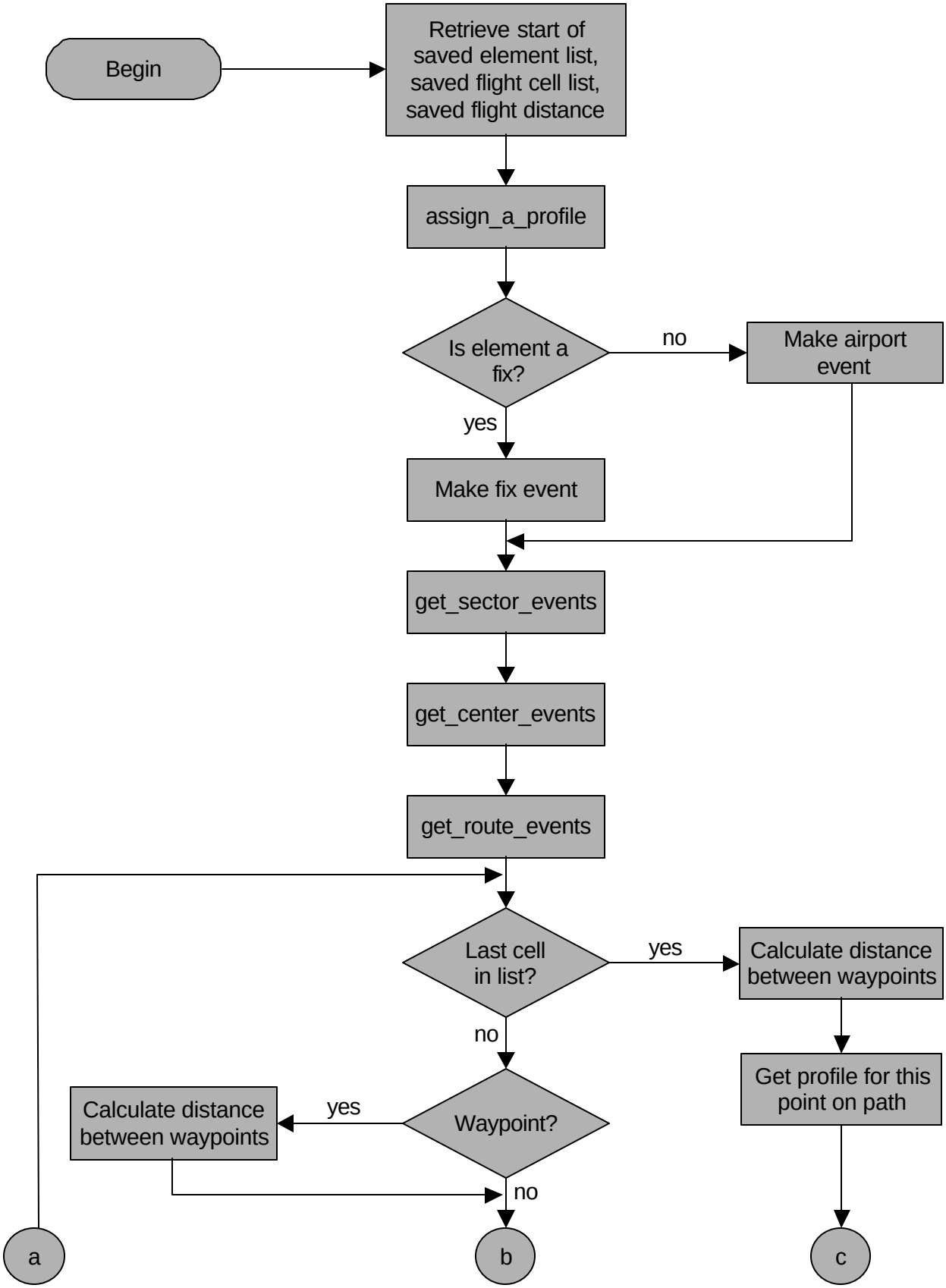


Figure 24-17. Sequential Logic for the make_event_list Routine

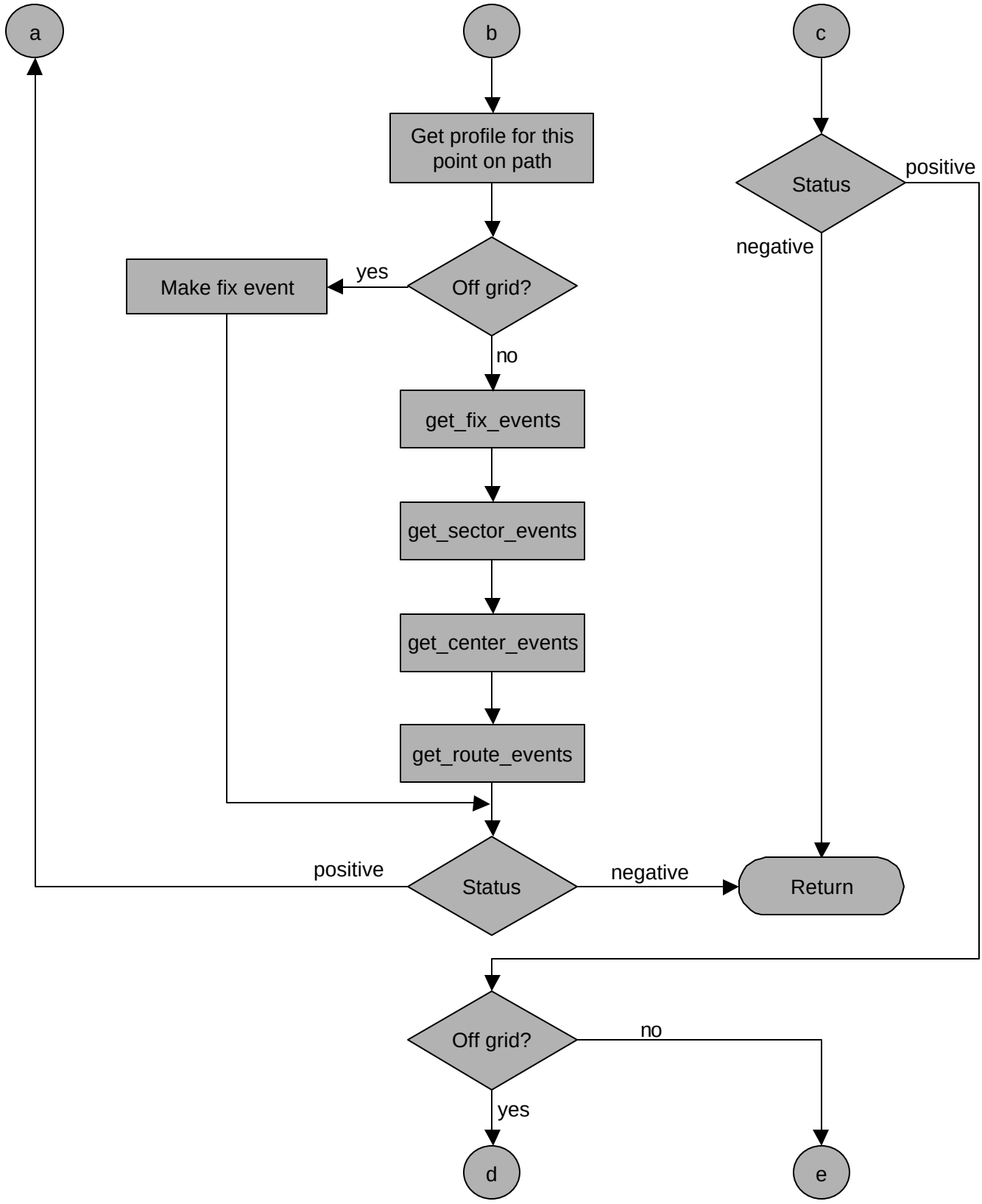


Figure 24-17. Sequential Logic for the make_event_list Routine (continued)

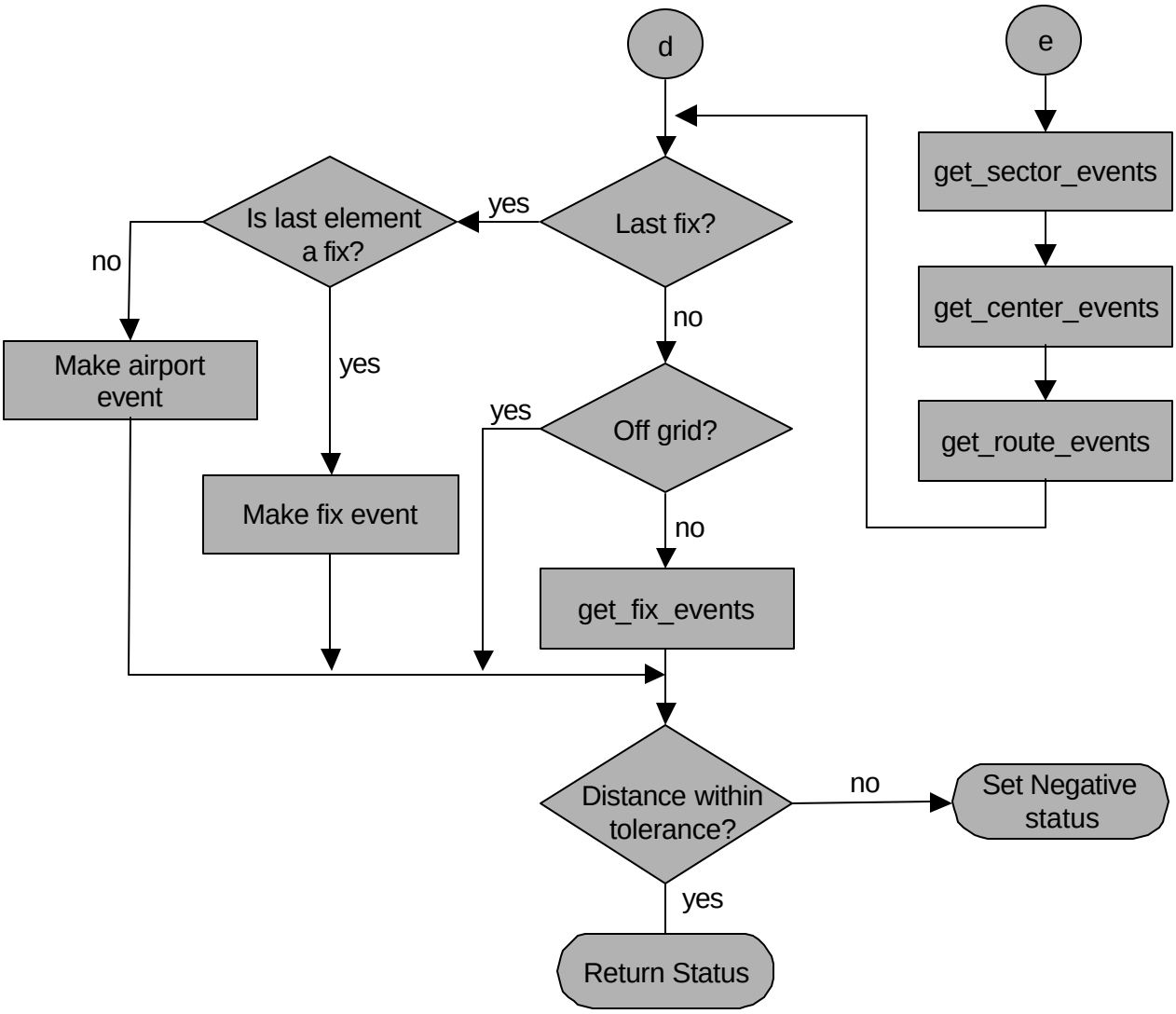


Figure 24-17. Sequential Logic for the make_event_list Routine (continued)

If there is more than one grid cell in the **cell list**, *make_event_list* calls *get_fix_events*, *get_sector_events*, *get_center_events*, and *get_route_events* to make any appropriate events for that grid cell. For all subsequent grid cells in the list, including the last one, the *get_profile_data* routine updates values in the **a_flight** record and calls *get_altitude_value* (Section 24.4.4.1), which is the routine that provides altitude and velocity data for each segment of the flight path.

For all grid cells between the first and the last, *make_event_list* continues to add distances between waypoints, make any fix event found in the **events_to_make** list that corresponds to the current cell, and make fix events for any **fixoffsets** found in the **cell list** itself. The latter are the offsets found at the waypoints in named routes, which were stored in the **cell list** when the paths along the routes were traced. When a specific fix is known, such as one saved in the **events_to_make** list or as a **fixoffset** in the **cell list**, that fix is added to the **event list**. When no specific fix is known, such as at a waypoint defined by an unnamed fix (such as a route intersection), *make_event_list* calls *get_fix_events.*

Table 24-10. Flight Record Data Structure

Flight Record							
Library Name: Profile_openlib			**Element Name:** Profile.h				
Data Item	**Definition**	**Unit**	**Legal Range**	**Var. Type**	**I/O by function+**		
					F_1	**F_2**	**F_3**
flt_id	flight identifier (e.g., AAL123)			array[1…10] of char	|	--	--
filed_fz_onground	indicates disposition of the filed Field 10 Field 10 filed on grnd=T Filed in air=F		T or F	Boolean	|	--	--
civ	indicates if the aircraft is civilian=T or military=F		T or F	Boolean	O	|	--
runaway	indicates that flight errors or inconsistencies are severe & fatal=T No major problem=F		T or F	Boolean	O	--	--
dsg_actual	actual designator taken from the FZ			array[1…4] of char	|	--	--
aircraft_index	index indicating a record in the Aircraft_Descriptor Map which describes the given flight		-1 to max_plane_type	short	O	--	--
dsg_index	index indicating the particular template aircraft assigned to this flight		1 to tot_templates	short	O	|	|
ascent_index	index indicating the particular ascent profile for this flight		1 to max_ascent_profiles	short	O	|	|
descent_index	index indicating the particular ascent profile for this flight		1 to max_descent_profiles	short	O	|	|
dist_total	total distance for this flight	nautical miles		INT32	|	|	|
origin_lat	latitude of the originating airport	radians		float	|	|	|
origin_lon	longitude of the originating airport	radians		float	|	|	|
dest_lat	latitude of the destination airport	radians		float	|	|	|
dest_lon	longitude of the destination airport	radians		float	|	|	|
dist_cruz	distance from the takeoff roll to the point at which the aircraft achieves cruising altitude	nautical miles		INT32	O	|	|
spd_cruz	cruising speed for this flight	(nautical miles/min) x 100		INT32	|/O	|	|
alt_cruz	cruising altitude for this flight	feet/100		INT32	|/O	|	|

+ **F_1** indicates *Assign_A_Profile*, **F_2** indicates *Get_Altitude_Values*, **F_3** indicates *Get_Time_Value*

Table 24-10. Flight Record Data Structure (continued)

Flight Record (continued)							
Library Name: Profile_openlib			**Element Name:** Profile.h				
Data Item	**Definition**	**Unit**	**Legal Range**	**Var. Type**	**I/O by function+**		
					F_1	**F_2**	**F_3**
dist_descent	distance from the begin descent point to the point where the aircraft touches down	nautical miles	0 to 160	INT32	O	|	|
previous.time	accumulated time from takeoff roll to the previous location of this flight	minutes		float	--	--	|
previous.lat	latitude at the previous location of this flight	radians		float	O	|	|
previous.lon	longitude at the previous location of this flight	radians		float	O	|	|
previous.dist	previous distance along the flight path	nautical miles		INT32	O	|	|
previous.phase	flight phase at the previous location for this flight			flight phase see Sec.15	O	|	|
previous.alt	altitude at the previous location for this flight	feet/100	0 to 600	INT32	O	|	|
previous.speed	speed at the previous location for this flight	(nautical miles/min) x100		INT32	O	|	|
now.time	flying time from the previous location to the current location	minutes		float	--	--	O
now.lat	latitude at the current location of this flight	radians		float	|	|	|
now.lon	longitude at the current location of this flight	radians		float	|	|	|
previous.dist	current distance along the flight path	nautical miles		INT32	|	|	|
now.phase	flight phase at the current location of this flight			flight phase see Sec. 15	O	O	|
now.alt	altitude at the current location for this flight	feet/100	0 to 600	INT32	O	O	|
previous.speed	speed at the previous location for this flight	(nautical miles/min) x100		INT32	O	|	|
no_descent	indicates whether or not the descent is modeled. ne_descent_modeled=T descent is modeled=F		T or F	Boolean	|	|	|
no_descent	indicates whether or not the flight must land on this call. Must land now=T, not land now=F		T or F	Boolean	|	|	|

+ **F_1** indicates *Assign_A_Profile*, **F_2** indicates *Get_Altitude_Values*, **F_3** indicates *Get_Time_Value*

If there are no fixes in the cell, but this is a waypoint on the flight path, an event is added to the event list for an unnamed fix (latitude/longitude fix) at the latitude and longitude of the center of the grid cell. *Make_event_list* calls *get_sector_events* and *get_route_events* to make events based on the altitudes retrieved in *get_altitude_value*. When the final grid cell is reached, first sector and route events are made, forcing a sector and route exit, if necessary. A fix event for any remaining fix on the **events_to_make** list or in the last grid cell is also made.

24.5.3.2.7.1 The get_fix_events Routine

Get_fix_events searches the grid cell to find the fix of the highest priority. The first priority is always given to any monitored fix in the cell. If no fix is monitored, priority is determined by fix type, with the types specified in priority order in the fix-priority file loaded during initialization. If there is no fix in the cell, no fix is returned.

24.5.3.2.7.2 The get_sector_events Routine

For each grid cell constituting the flight route, *make_event_list* invokes *get_sector_events* once.

Get_sector_events notes what, if any, changes have occurred in the ARTCC designation from the last grid cell to the current grid cell. It takes no action if there has been no change in the designation. If there has been change, it takes action in accord with the following Table 24-11 Sector Event Logic.

Table 24-11. Sector Event Logic

Sector Event Logic		
Former Cell	**Current Cell**	**Action**
no entry	no entry	None
no entry	sector K	Create entry event to sector K
sector K	sector K	None
sector J	sector K	Create exit event from sector J, & entry event to sector K
sector K	no entry	Create exit event from sector K

A sector event is determined by the identification of sector type at the current altitude of the flight path from the current grid cell (i.e., a superhigh sector when the flight is at a superhigh level, a high sector when the flight is at the high level [or the superhigh level where no superhigh sector exists], or a low sector at the low altitude level). The sector thus retrieved is first checked to ensure that the actual altitude of the flight at that point is not below the bottom altitude for that sector. If not, the sector is checked against the last sector the flight was in, which is identified in the prevsect field of the route_context record. If the two sectors differ, an exit event is added to the event list for the previous sector, if any, and an entrance event is added for the current sector, if any.

24.5.3.2.7.3 The get_center_events Routine

For each grid cell constituting the flight route, *make_event_list* invokes *get_center_events* once.

Get_center_events notes what, if any, changes have occurred in the ARTCC designation from the last grid cell to the current grid cell. It takes no action if there has been no change in the designation. If there has been change, it takes action in accordance with the following Table 24-12 ARTCC Event Logic.

Table 24-12. ARTCC Event Logic

ARTCC Event Logic		
Former Cell	**Current Cell**	**Action**
no entry	no entry	None
no entry	center K	Create entry event to center K
center K	center K	None
center J	center K	Create exit event from center J, & entry event to center K
center K	no entry	Create exit event from center K

NOTE: If the current grid cell is the first of the flight route, the former cell will be interpreted as a No Entry. Also, on-grid cells which are not assigned to any ARTCC have an ARTCC designation of No Entry. Generally, these cells are over the oceans near the U.S. coast or over Mexico or Canada. Alaska constitutes an ARTCC but is not part of the grid. *Get_center_events* does not create events for Alaska.

If no action is required, program control returns directly to the invoking routine, *make_event_list.*

In the cases where the ARTCC event logic warrants the creation of an entry or exit event, the routine assigns appropriate values to certain parameters, and using them it initiates the event creation process by invoking the routines *make_event* followed by *add_to_evlst.* It then returns to *make_event_list.*

Subordinate routines define parameters such that the event to be created has the correct properties. These properties specify the ARTCC event type, its status as neither monitored nor as a waypoint, an index identifying the particular ARTCC, and whether the event corresponds to an entry or an exit from a ARTCC. They also take the parameters associated with the aircraft flight profile at the current grid cell and put them into an event list record at the proper location. The parameters characterize the flight's state at the time of event creation specifying such items as aircraft altitude and velocity.

No special action is taken upon flight termination. The final ARTCC is not exited simply because the aircraft lands.

24.5.3.2.7.4 The get_route_events Routine

For each grid cell constituting the flight route, *make_event_list* invokes *get_route_events* once. *Get_route_events* determines which, if any, jet or Victor routes the flight enters or exits, and it creates events for the event list to indicate these entries and exits. The logical flow of the *get_route_events* processing is shown in Figure 24-18.

Get_route_events accesses the **grid database** information stored for each grid cell for the characteristics of that particular geographical location. Any named routes passing through or terminating in the grid cell are listed in the data set associated with the cell. *Get_route_events* examines each grid cell in turn, collecting the names of any jet routes or Victor routes in the cell, and adds those names to a list of current potential routes. **Currentlist**, an array of records, provides a record for each new route name encountered. Each record contains a field holding the current number of times that route has been sighted during the route processing.

Currentlist also stores information specifying the latitude, longitude, altitude, phase, and velocity of the proposed route entry and predicted exit. Ultimately, the predicted exit is subsumed by the actual exit. *Get_route_events* estimates entry and exit points in terms of relative location within the flight's event list, and it updates the estimates as the flight proceeds. It stores whatever additional information is necessary to construct an entry or exit event with the appropriate parameters in the correct sequence.

If a route remains unsighted for a space of several grid cells, the route name is dropped from **currentlist**. If sightings accumulate enough to push the value across a threshold, the routine creates a route entry event. When a route for which an entry event has been created fails to be seen for several grid cells, the routine creates a route exit event.

The decision to create an entry (or exit) event occurs several grid cells after the route first (or last) appears. *Get_route_events* therefore sets aside for the event creation the proper parameters (such as altitude and latitude) associated with the actual entry or exit point. Other kinds of events (such as a sector crossing) may intervene between the point the route entry or exit actually occurred and the point at which the *get_route_events* logic becomes certain it really happened. *Get_route_events* therefore inserts the route event anywhere in the event list rather than appending it to the end of the list as is done in the other event creation routines.

Get_route_events uses several embedded subroutines in its processing. Upon successful completion of its processing, *get_route_events* returns the number of route events created back to the invoking routine, *make_event_list*.

Get_route_events begins with some initializations including a check to see if a new flight has begun. If a new flight has begun, the routine sets appropriate initial values for the various fields of the **currentlist** records. It does this to prevent leftover route information from a prior flight from mingling with that generated for the current flight. If the flight is not a new one, *get_route_events* detects that fact and continues without reinitializing.

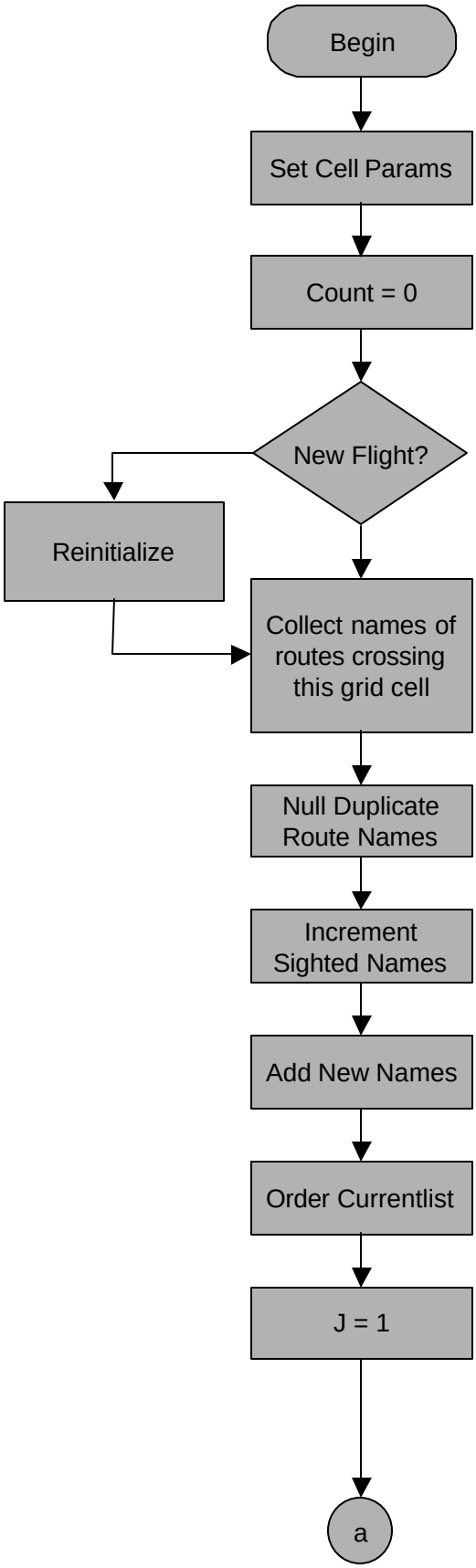


Figure 24-18. Sequential Logic for the get_route_events Routine

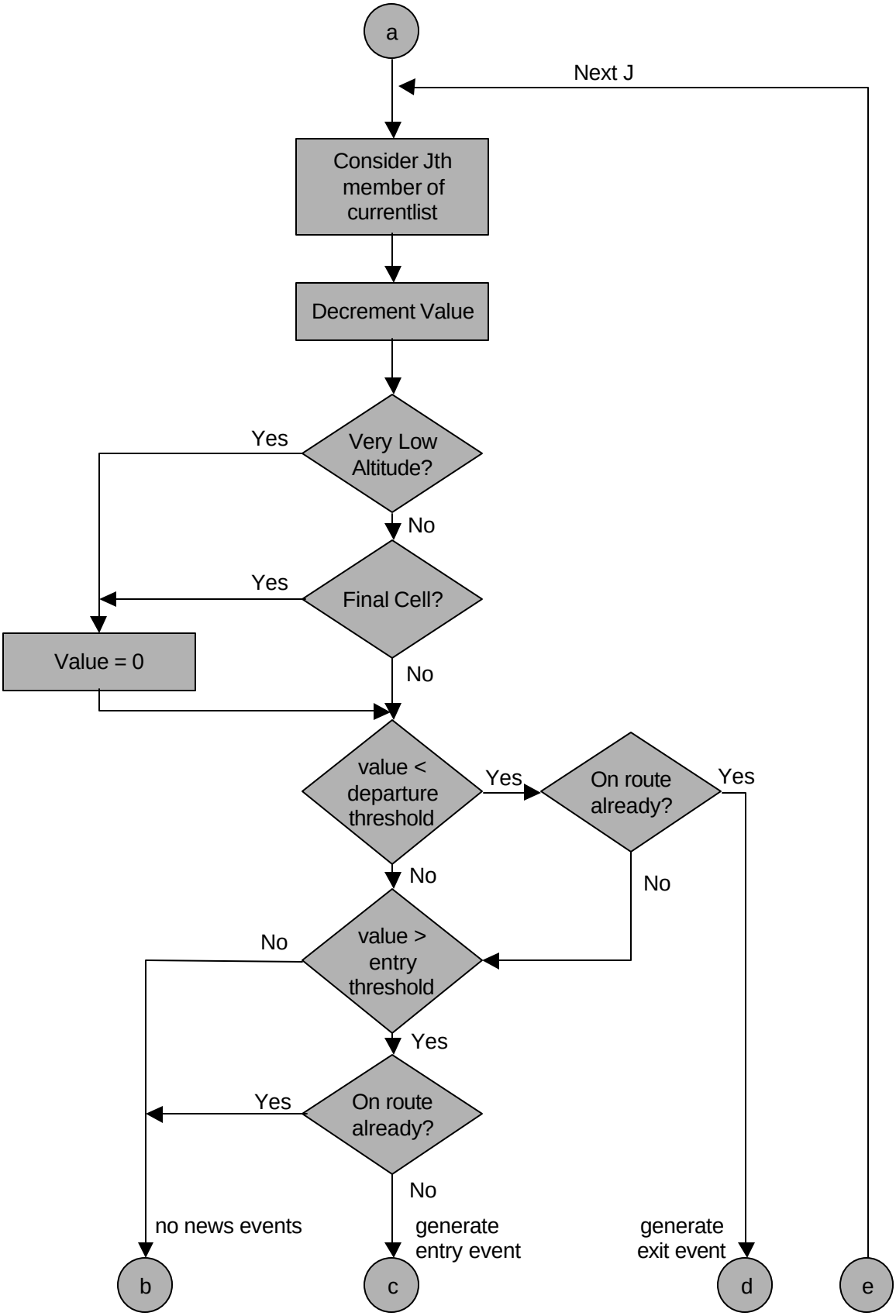


Figure 24-18. Sequential Logic for the get_route_events Routine (continued)

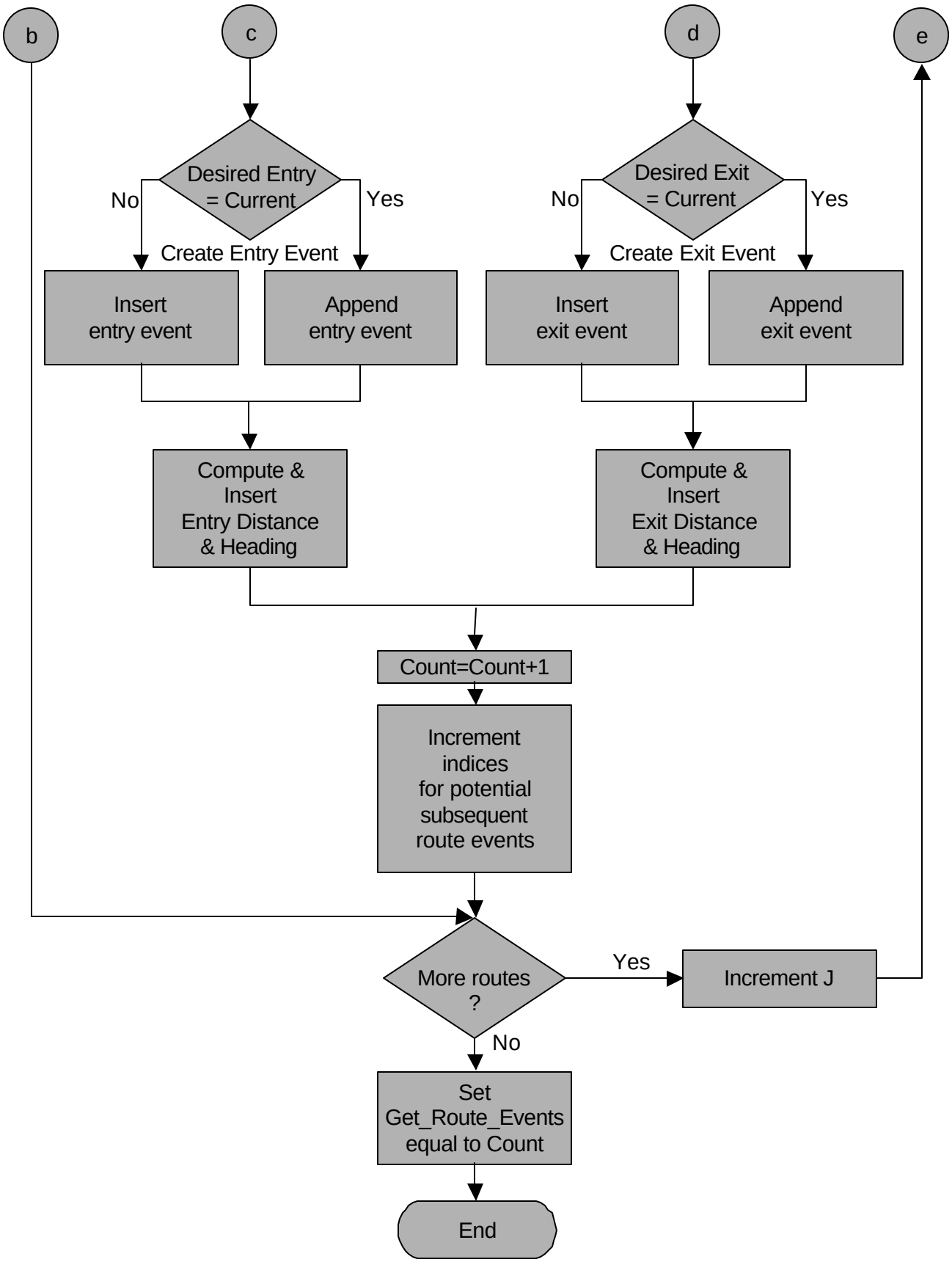


Figure 24-18. Sequential Logic for the get_route_events Routine (continued)

Get_route_events notes which jet route and victor routes appear in the particular grid cell being traversed, and places the names of those routes into the array **thiscellsnames**. It sorts the names in the cell with the discriminant of allowable altitude on the different route types, making use of the particular flight's altitude. Victor routes occupy altitudes below 18,000 feet. Jet routes span 18,000 to 45,000 feet including the end points. Of the various types of named routes, *get_route_events* actively considers only jet routes and Victor routes.

> **NOTE:** *Get_route_events* could handle other route types. Only one subordinate embedded routine, namely *collect_cell_names,* requires modification, if other types of routes are to be considered.

Get_route_events ensures that a route name is only added once to the **currentlist.** Some grid cells (especially at airports) duplicate route names, necessitating this action. *Get_route_events* compares the route names appearing in t**hiscellsnames** with those residing in **currentlist**. When it sights a name on this grid cell already in **currentlist**, it increments it in **value**. Also, it updates some of the other fields of the **currentlist** record to reflect the flight's current parameters such as latitude and altitude. It then takes any route names which appear in **thiscellsnames**, but which are not already in **currentlist**, and it creates for them a record in **currentlist**. That record contains fields for the route name, its value, and some flight parameters.

At this point, *get_route_events* starts a loop in which it considers one by one each of the names presently in **currentlist**.

For the first action in the loop, *get_route_events* decrements the **value** field for each (if any) current route name's record. This ensures that routes will ultimately reduce in value to zero and be deleted unless something occurs to offset the periodic, grid cell-by-grid cell, decrementing.

> **NOTE:** The incrementing of value occurs previous to the loop, as described above.

Prior to beginning actual event creation, the routine takes some special actions to ensure proper handling of flight termination. If the altitude falls below 1000 feet at the current grid cell, flight termination logic assigns a **value** of zero to each route name in **currentlist**. The routine's logic will subsequently close out any open routes with **value** of zero. This action ensures that all the named routes associated with a flight path are closed out upon landing. If the grid cell at hand is the final one of the linked list of cells constituting the path, the flight termination logic assigns a **value** of zero to each route name in **currentlist**. The routine's logic will then close out any open routes with **value** of zero. This ensures that all named routes are closed upon the completion of an event list for a flight, regardless of whether the flight descends for landing, or whether the event list is ended for other reasons.

Actual event creation begins with a check of each route name in **currentlist** to detect whether any names' **value** has exceeded the entry threshold for creating an entry event. Event creation logic checks the name's **currentlist** field **onflag** to ensure that the route has not already had an entry event. If a route entry event already exists, the logic ignores the particular named route. If the names' **value** exceeds the entry threshold and **onflag** has been false, the routine creates an entry event.

The event creation logic examines the desired event position, which had been stored in **currentlist**, and compares it with the current event list position, **evpos**. If the proposed new entry event would be the most recent event, the logic appends the appropriate event to the end of the

event list. If the desired event position is at some position earlier than that of the current event list pointer, the logic inserts the appropriate event at the proper position in the list. For either appended or inserted events, event creation uses the appropriate parameters from the grid cell at which the route entry was first noted. These parameters include latitude, longitude, altitude, phase, and velocity. **Currentlist** holds them.

The logic for exit events works in a manner similar to the logic for entry events. The procedures *append_exit_event*, and *insert_exit_event* carry out roles analogous to their entry counterparts. The logic triggers exit event creation when a **currentlist's** name's **value** declines below the exit threshold. The logic does not attempt to exit any route that has not been entered. The logic checks **onflag** to verify whether the flight has entered the route.

The event list entries include distance and bearing relative to the preceding event. When *get_route_events* creates an event, it computes distance and bearing. *Get_route_events* takes special care with the cases where the event is inserted into the event list rather than appended to its end. If inserted, *get_route_events* recomputes distance and bearing for any events which follow the newly created route event.

Error Conditions and Handling

The *make_event_list* has four types of fatal errors, each of which is identified by a status code message indicating the type of error to the *Semantic Parser*. The messages are explained below:

(1) "Error in profile data" — input data to aircraft profile routines (such as cruising altitude and velocity, aircraft identifier) was too inaccurate to reconstruct

(2) "Inaccurate distance" — total flight path distance as initially calculated, and later determined in second pass through flight path, varied by more than 15 nautical miles

(3) "Event list full" — the maximum number of events allowed has been exceeded

(4) "Distance too long for aircraft type" — flight distance too long for the aircraft used

24.5.4 The Flight Profiler Module

Purpose

The purpose of the *Flight Profiler* is to determine phase, speed, and altitude of an aircraft at any time during the flight.

Input

Input to the *Flight Profiler* is the **flight record** shown in Table 24-10.

Output

Output from the *Flight Profiler* is an updated **flight record**.

Processing Overview

The *Flight Profiler* is mainly comprised of the two routines described on the following pages.

24.5.4.1 The get_altitude_value Routine

Purpose

The purpose of this procedure is to determine the current flight phase, speed, and vertical position of a flight when given the current distance along the horizontal flight path as well as the previous state of the flight.

Input

The input is passed in the **flight record** and is described in Section 15. A character **I** in the second from the last column in the **flight record** table (see Table 24-10) indicates that the data item is used as input in this procedure. The characters **I/O** indicate that the item is initially an input parameter and may be altered by the procedure to function later as an output parameter.

Output

The output is passed in the **flight record** and is described in Section 15. A character **O** in the second from the last column in the **flight record** table (see Table 24-10) indicates that the data item is used as output in this procedure. The characters **I/O** indicate that the item is initially an input parameter and may be altered by the procedure to function later as an output parameter.

Processing

Several important parameters along with the previous flight state determine the current altitude, speed, and flight phase.

The logic used in this procedure is based primarily on the value of the previous flight phase for the flight, and secondarily on the current distance along the flight path, **dist_cruz** (i.e., during the ascent, the distance along the flight path at which point the flight starts to level out), and **dist_down** (i.e., the distance along the flight path at which point the flight starts to descend). Tertiary factors include such information as the current lat-lon, the origin lat-lon, the total flight distance, **get_down** (i.e., a boolean input parameter that indicates whether the aircraft is in the same grid cell as the destination airport), etc.

The flight phase value (see Section 15) at the previous aircraft location is used to determine the possible current state according to the following transition rule. The current flight phase may assume the value of the previous flight phase or any higher value with one exception: a flight with a previous value of **climb** cannot take on the current value of **level-out** as diagrammed in Figure 24-19.

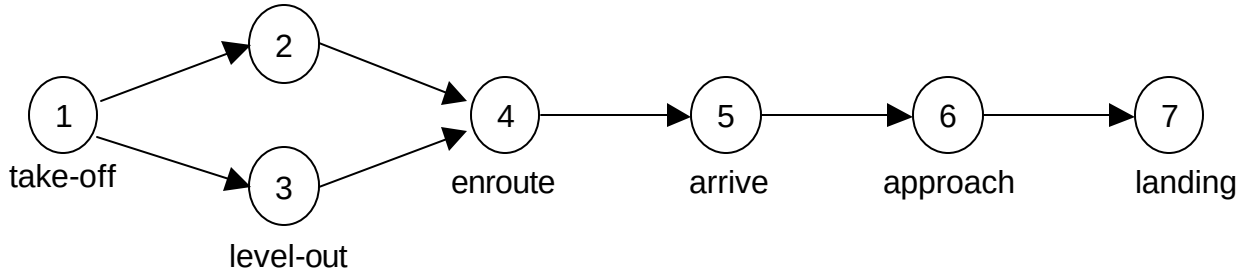


Figure 24-19. Possible Transitions from the Previous Phase to Current Phase

While the previous phase limits the possibilities, the current distance along the flight path is then used to determine the vertical flight orientation (i.e., ascending, flying level, or descending). The current distance is compared against the value of **dist_cruz** and/or **dist_down**. Assuming that all orientations are possible, the following comparison is applied: If the current distance is less than the **dist_cruz** value, the flight is ascending: if it is greater than **dist_cruz** and less then **dist_down**, the flight is now flying level; if greater than **dist_down**, it is descending.

After determining the vertical orientation, *get_altitude_value* considers various criteria to determine the current flight phase. When the flight is ascending, the altitude and speed are looked up in the **ascent_by_dist** map. The flight phase is then determined by the current altitude; if less than FL100 the phase is set to **take-off**, otherwise the phase is **climb**. When the flight is flying level, the altitude is set to the cruise altitude. The flight phase is determined by the straight distance between the origin airport and the current position; if it is less than 30 nautical miles, the phase is set to **level_out** (i.e., the aircraft is within the origin TCA); otherwise the phase is **enroute**. The speed is set to cruise speed for en route flying but is adjusted under FAA speed limits within the TCA. When the flight is descending, the altitude and speed are looked up in the **descent_by_dist** map. The flight phase is determined by the altitude and the value of **get_down**: if it is greater than FL120, the phase is **arrive**; if it is less than FL120, the phase is **approach**; and if **get_down** is true, the flight phase is set to **landed**. Figure 24-20 describes the *get_altitude_value* routine in more detail.

Error Conditions and Handling

Any prospective errors are handled by the *assign_a_profile* routine as described in section 24.4.4.2.

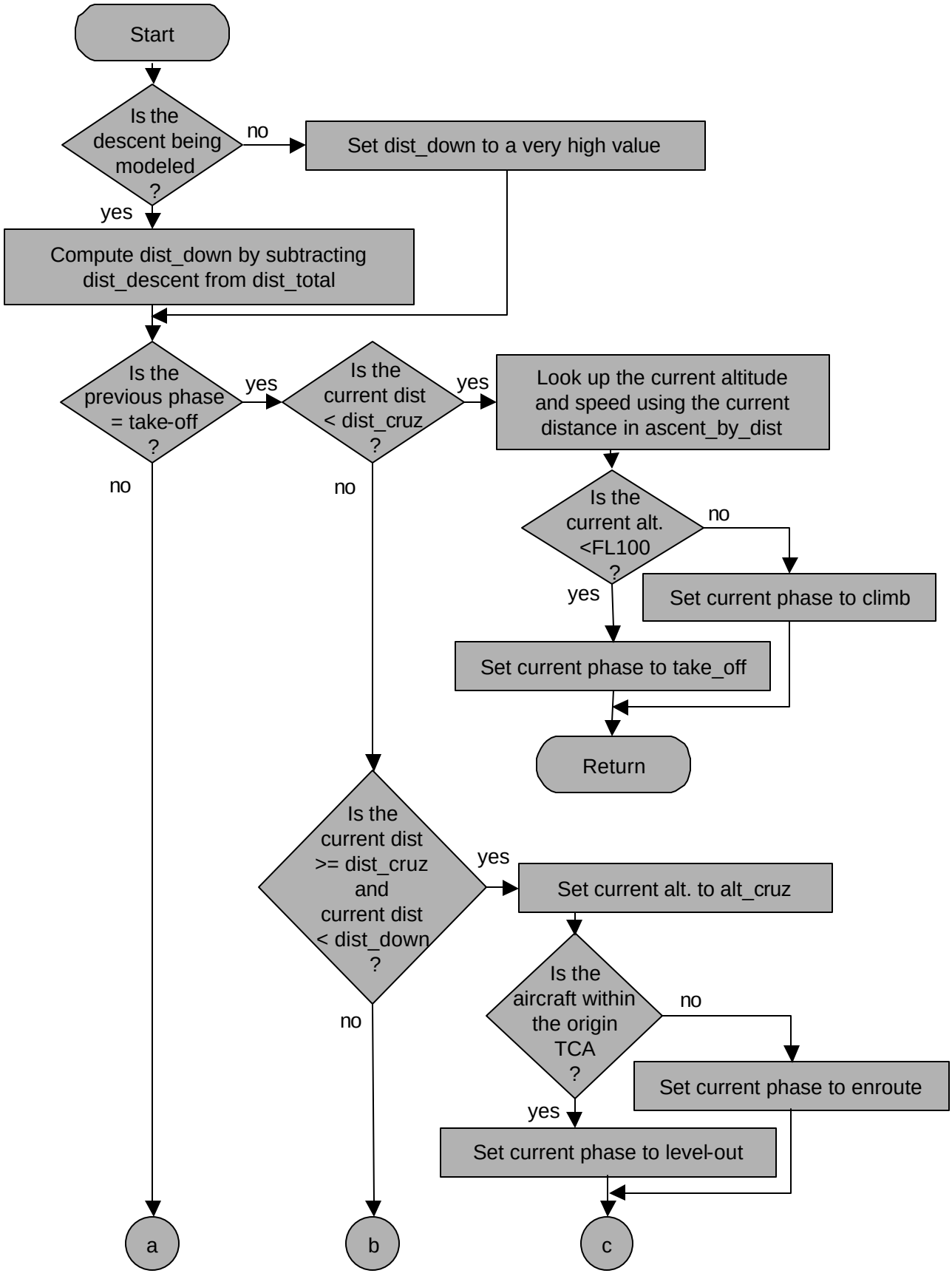


Figure 24-20. Sequential Logic for get_altitude_value Routine

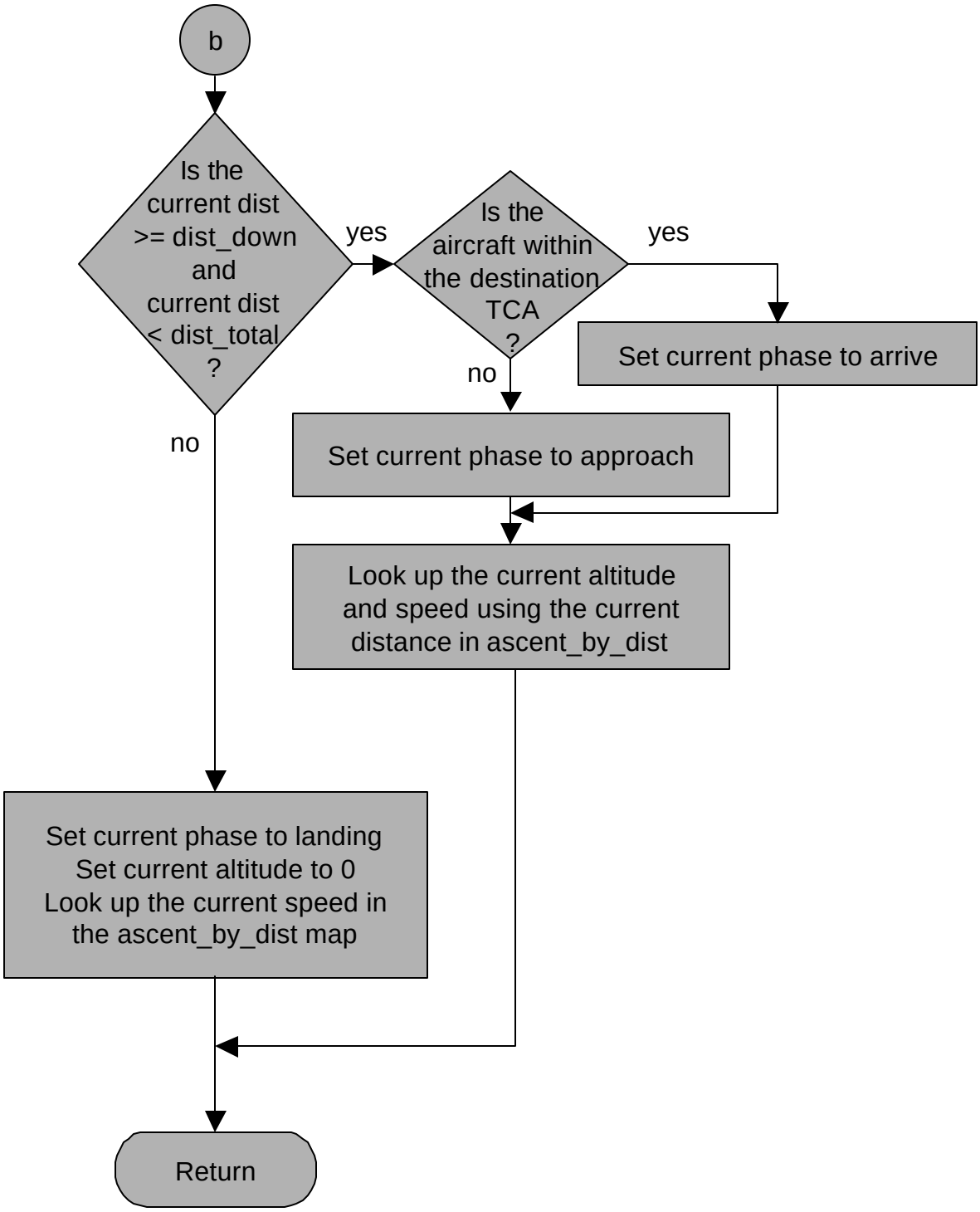


Figure 24-20. Sequential Logic for get_altitude_value Routine (continued)

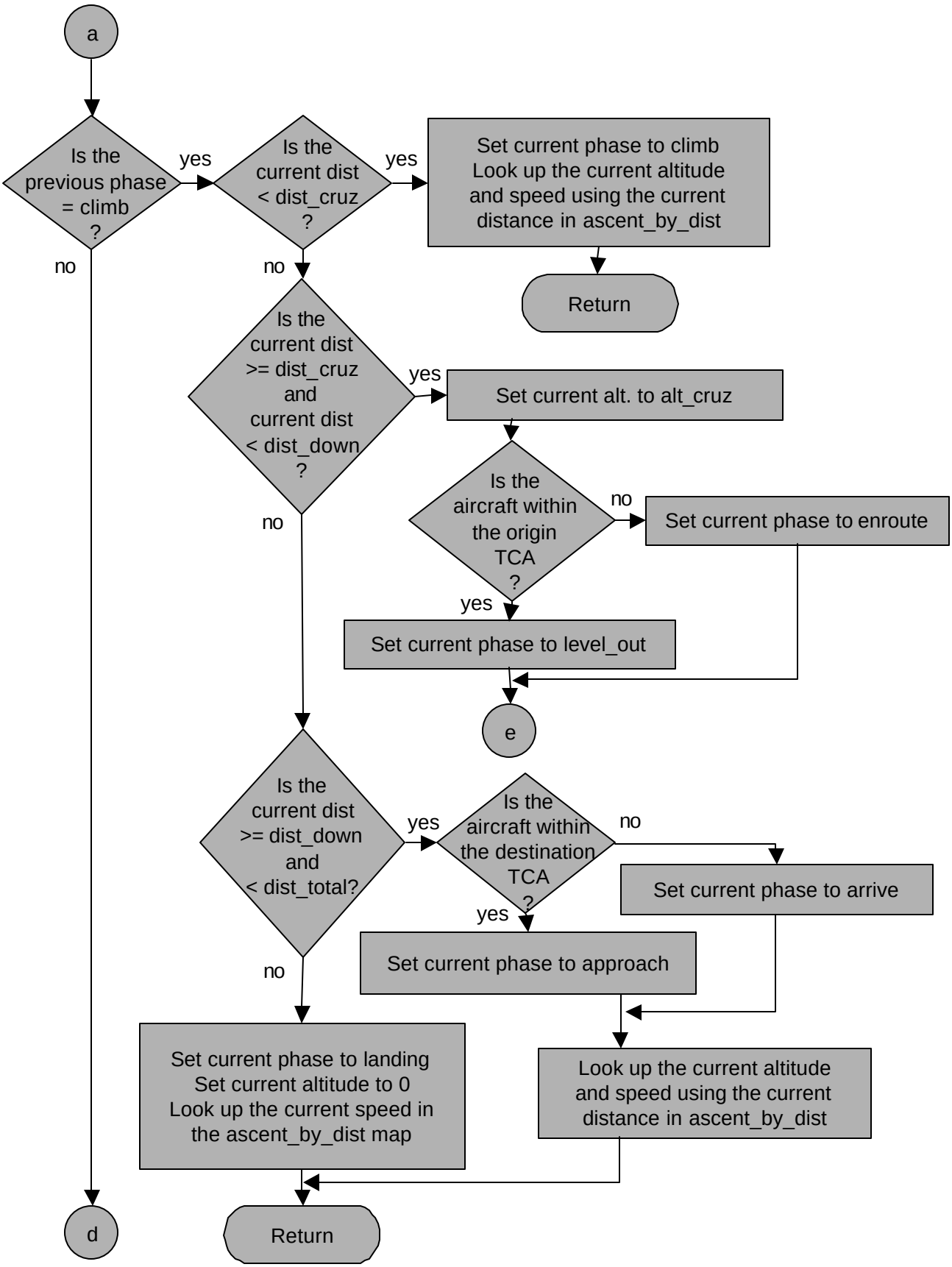


Figure 24-20. Sequential Logic for get_altitude_value Routine (continued)

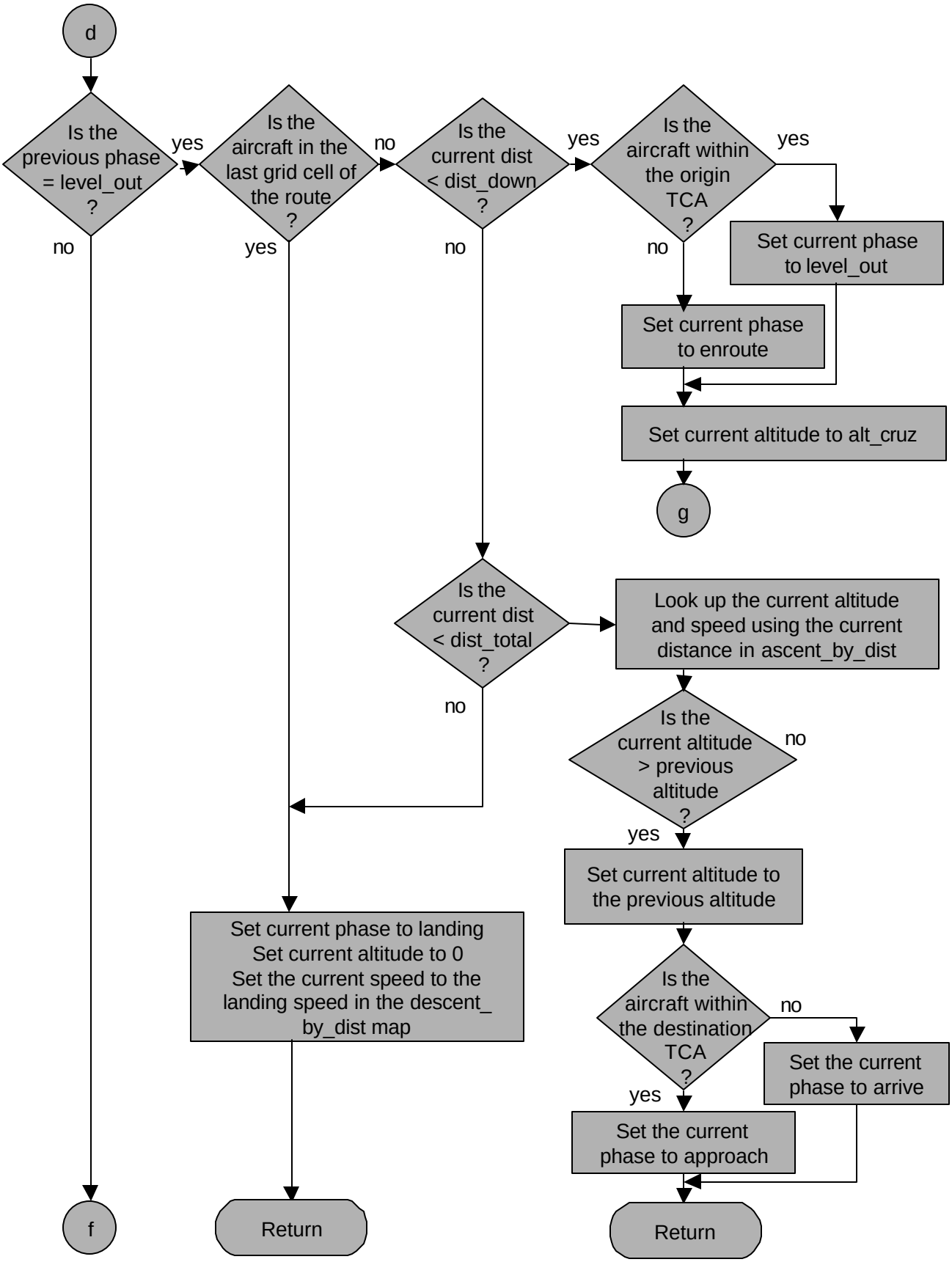


Figure 24-20. Sequential Logic for get_altitude_value Routine (continued)

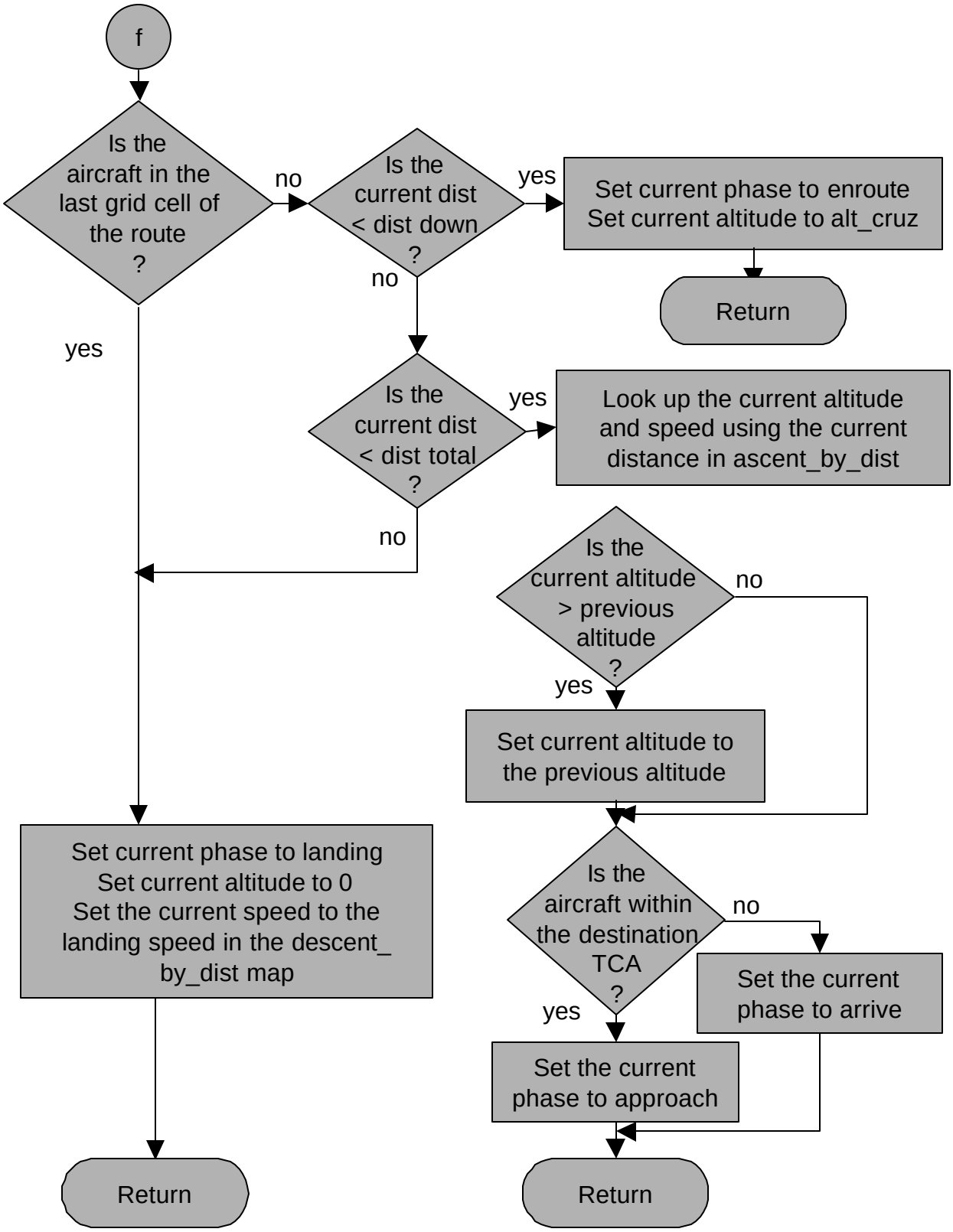


Figure 24-20. Sequential Logic for get_altitude_value Routine (continued)

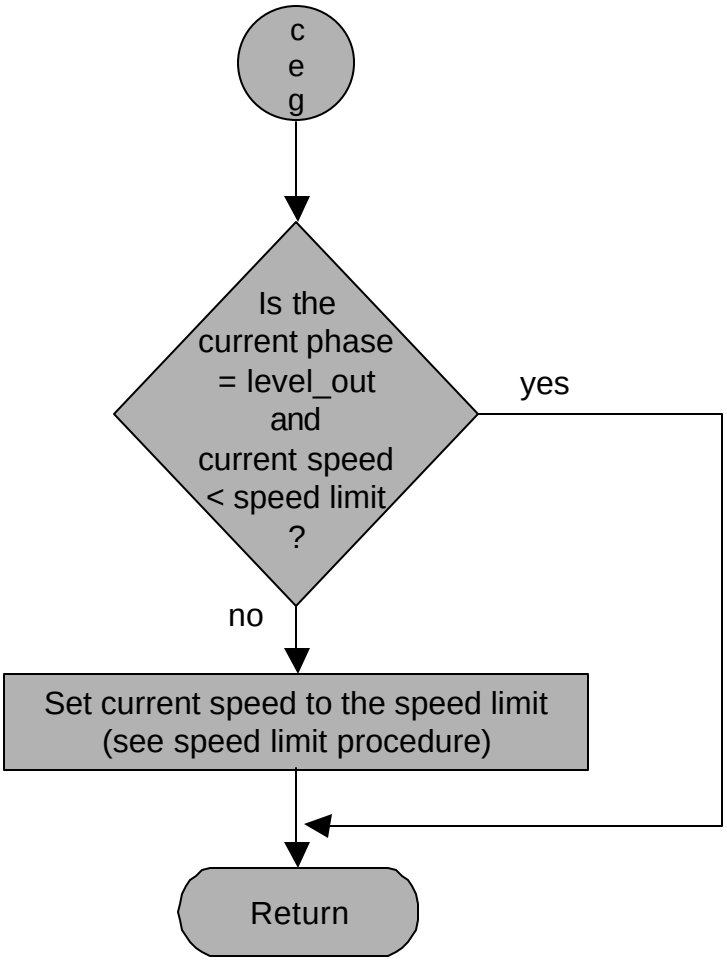


Figure 24-20. Sequential Logic for get_altitude_value Routine (continued)

24.5.4.2 The assign_a_profile Routine

Purpose

This procedure has two purposes: to check for inconsistencies and errors in the input values of the flight record (see Section 15), and, if the data are sufficiently consistent, to initialize the profile parameter values. These values will be used by later profile procedures to compute flight phase, altitude, speed, and time values for the flight.

Input

The input is passed in the **flight record** and is described in Section 15. A character **I** in the third from the last column in the **flight record** table indicates that the data item is used as input in this procedure. The characters **I/O** indicate that the item is initially an input parameter and may be altered by the procedure to function later as an output parameter.

Output

The output is passed in the **flight record** and is described in Section 15. A character **O** in the third from the last column in the **flight record** table indicates that the data item is used as output in this procedure. The characters **I/O** indicate that the item is initially an input parameter and may be altered by the procedure to function later as an output parameter.

Processing

The sections below describe the major tasks involved in assigning an ascent and descent profile and establishing profile parameter values for a flight. A flowchart is presented in Figure 24-21 to show the sequential logic for this procedure.

Characterize the Aircraft. As described in the Aircraft Dynamics Modeling section (see Section 23) there are several levels in characterizing an aircraft. At a gross level there are seven **grp** categories which include pistonprops, turboprops, large commercial jets, etc. Each has a relatively wide performance envelope. At a lower level there are 44 aircraft templates, and each is associated with a **grp** category. Each template is a popular aircraft model and has a narrower performance envelope than those associated with the **grp** categories. An aircraft template is associated with one or more ascent profiles, each of which is dependent on the total distance of the flight. At the lowest level a designator describes a specific aircraft model.

Currently, there are 800 designators described in the **Aircraft_Descriptor** map, although there are more which have not yet been included. The record fields of the map show that each designator is described in terms of its **grp** category and the index value of its template aircraft.

The aircraft used in the flight is characterized through the use of a binary search algorithm that matches the input designator value (i.e., **dsg_actual**) against the designator values in the **Aircraft_Descriptor** map (i.e., **dsg**). When a correct match is made, the template value and **grp** value can be used, in part, to determine the ascent and descent index, respectively. If the value of the input designator cannot be found, a best guess procedure is employed to characterize the aircraft used in the flight.

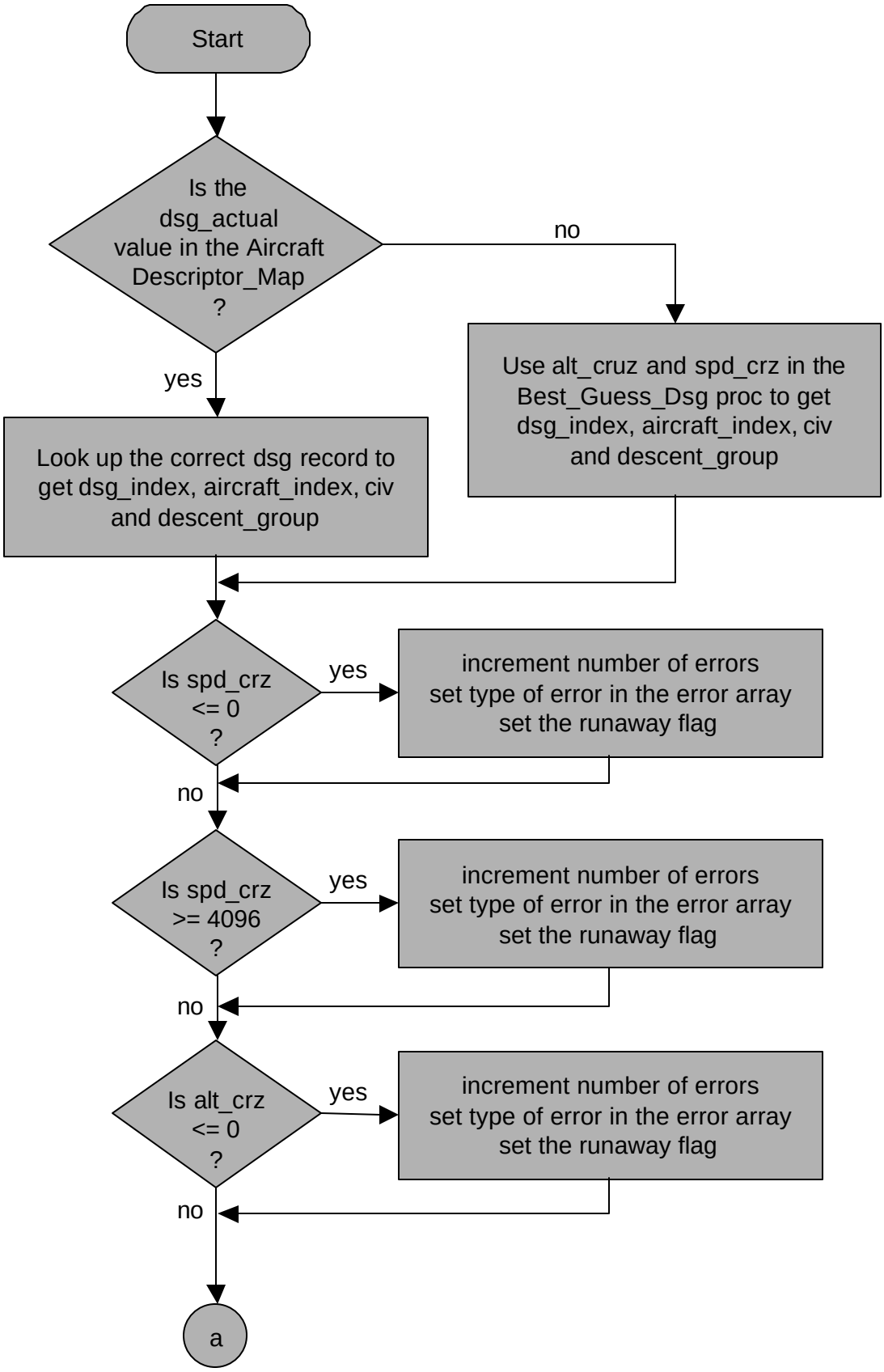


Figure 24-21. Sequential Logic for assign_a_profile Routine

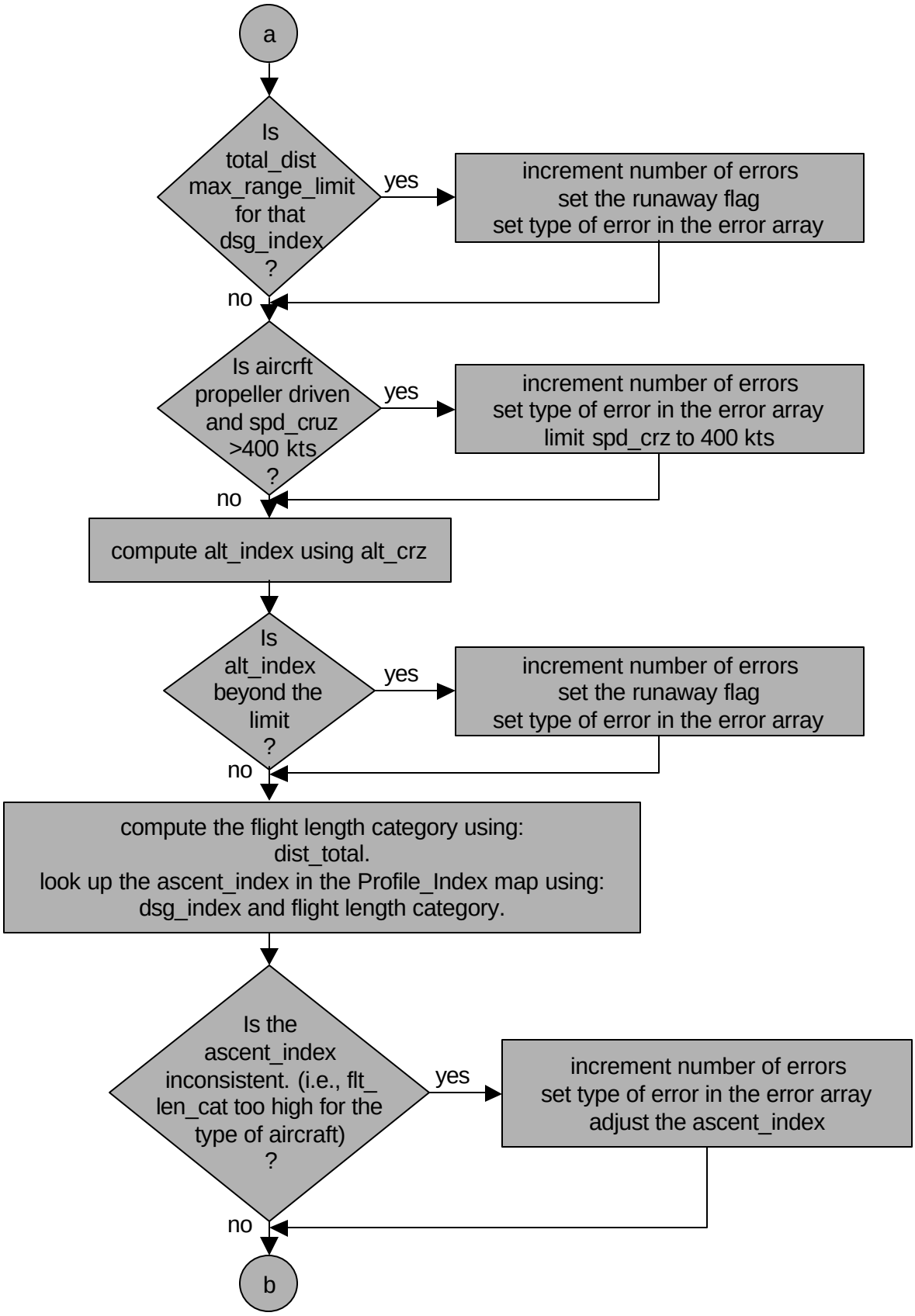


Figure 24-21. Sequential Logic for assign_a_profile Routine (continued)

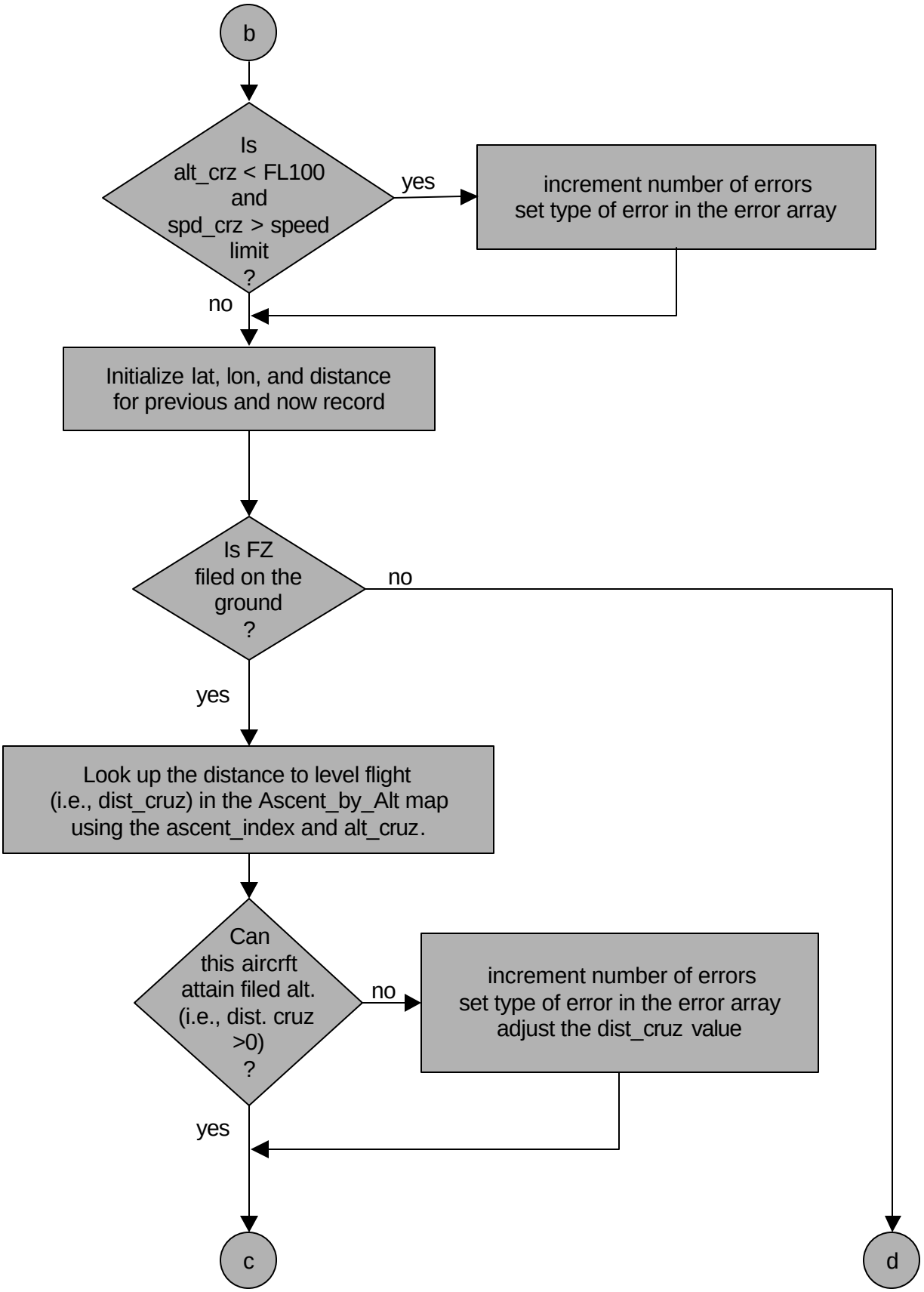


Figure 24-21. Sequential Logic for assign_a_profile Routine (continued)

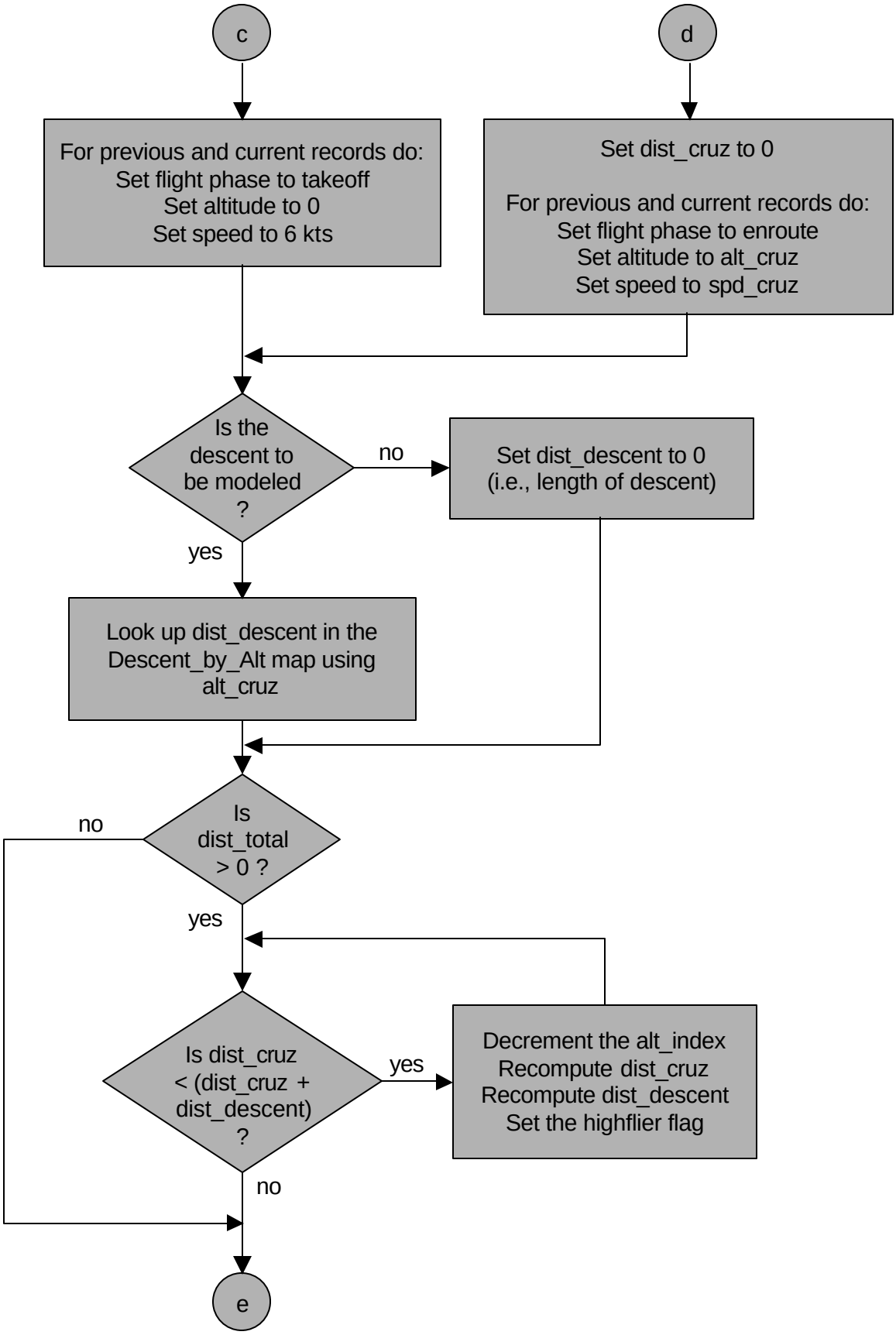


Figure 24-21. Sequential Logic for assign_a_profile Routine (continued)

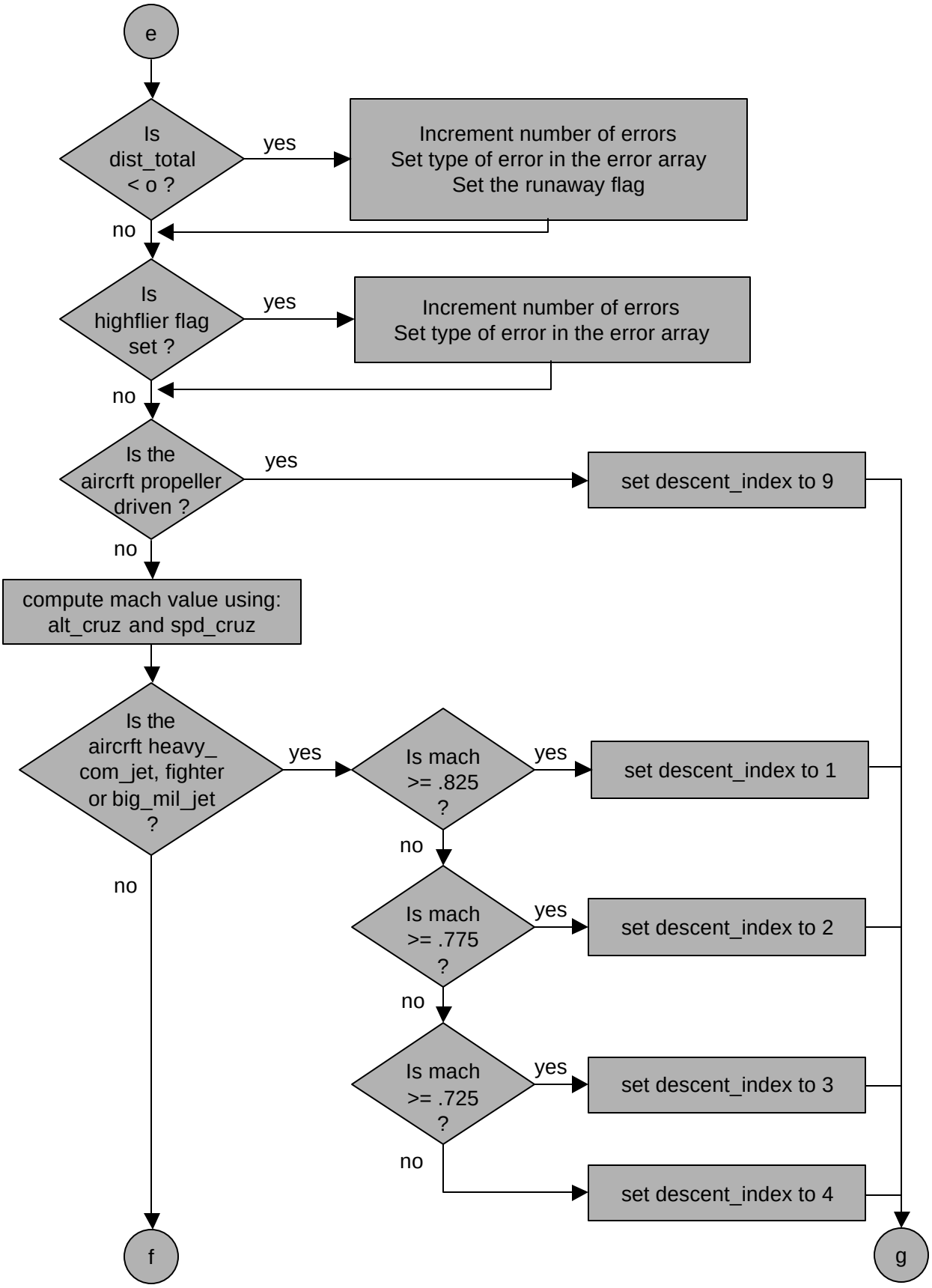


Figure 24-21. Sequential Logic for assign_a_profile Routine (continued)

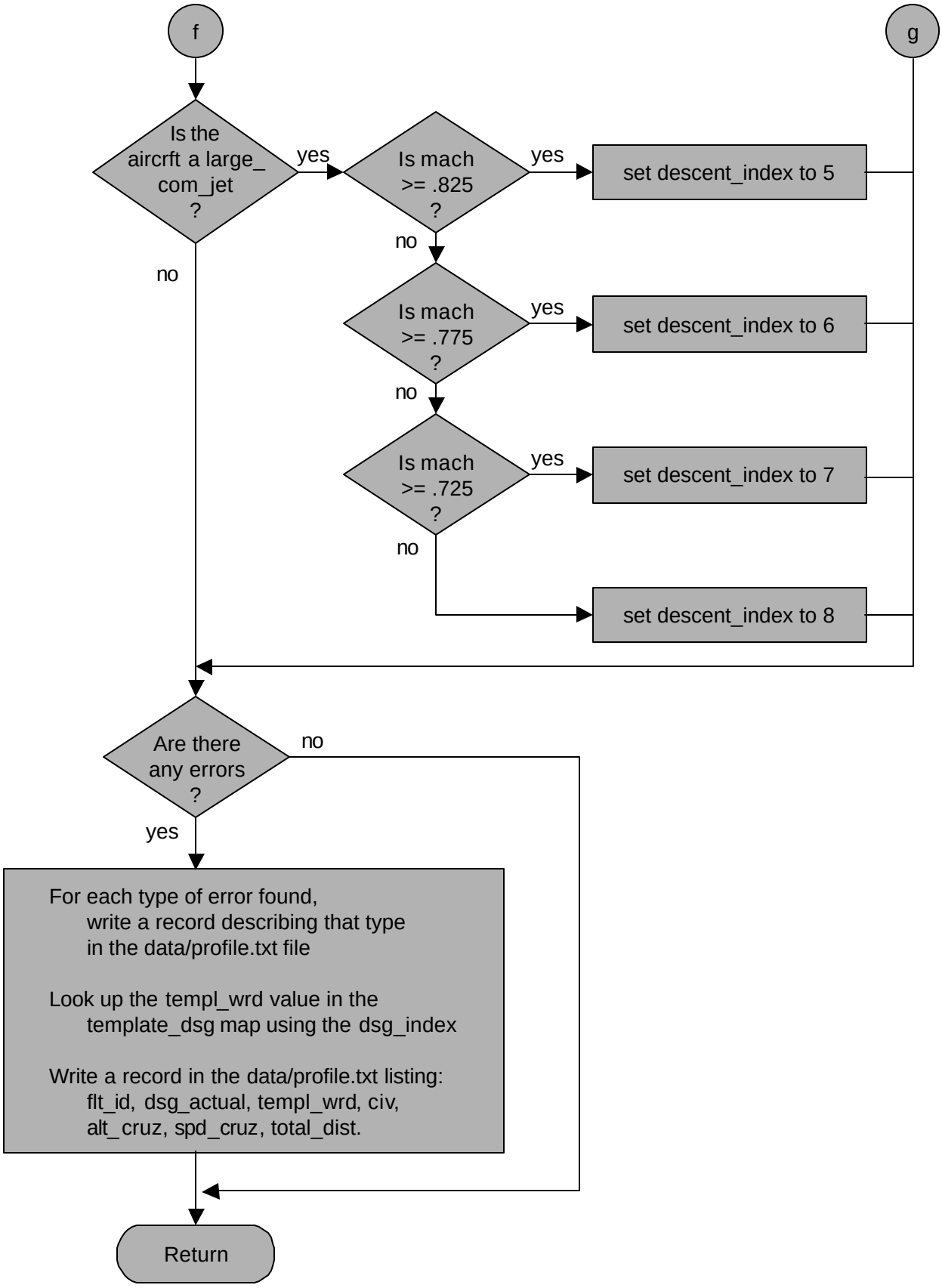


Figure 24-21. Sequential Logic for assign_a_profile Routine (continued)

When the flight designator cannot be found in the **Aircraft_Descriptor** map, the cruising speed and altitude values are used to guess an appropriate template aircraft value. Based on Figure 1-1 in "Design for the National Airspace Utilization System" (FAA SRDS, June 1962), a simplified set of disjoint performance envelopes were established. The boundaries of the envelopes may be comprised of constant altitudes, indicated airspeeds or mach numbers, and each of these boundary constructs is a function of the cruising speed and altitude. Also, each of the seven envelopes is associated with a **grp** and template value. Thus, the cruising speed and altitude of a flight matches one to one with a template-grp combination. The best guess matching procedure is illustrated in Figure 24-22.

Prepare Ascent Parameters. In this task the flight is assigned an ascent profile. An ascent profile mathematically describes the take-off and climb trajectory of the flight. For each ascent profile there are two relationships, which are inversely related to each other via a table: given the distance along the flight, one can find the altitude; conversely, given the altitude, one can find the distance, speed, or time along the flight.

There are 128 ascent profiles; each is associated with an aircraft template and total flight distance range combination. *Assign_a_profile* assigns the ascent profile by first determining one of seven flight distance categories using the *categorize_length* procedure. The index of the template aircraft (**dsg_index**) along with the distance category value are used to look up the **ascent_index** value in the **Profile_Index** map.

If the flight's take-off is to be modeled, the distance along the flight path to the leveling out point (i.e., **dist_cruz**) must be computed. *Assign_a_profile* calculates this parameter by computing a cruising altitude index and using that to look up the **dist_cruz** value from the as**cent_by_alt** map. Also, the sub-record fields of the flight record (see Section 15) are set appropriately for take-off (e.g., **altitude** = **0**, **speed** = **10**, **phase** = **takeoff**). If the flight take-off is not to be modeled, the **dist_cruz** value is set to zero and the sub-record fields are set for en route flight.

Prepare Descent Parameters. In this task the flight is assigned a descent profile. A descent profile mathematically describes the arrival and approach trajectory of the flight. As described in Section 15, for each descent profile there are two relationships which are inversely related to each other via a table: given the distance along the flight, one can find the altitude; conversely, given the altitude, one can find the distance, speed, or time to the landing.

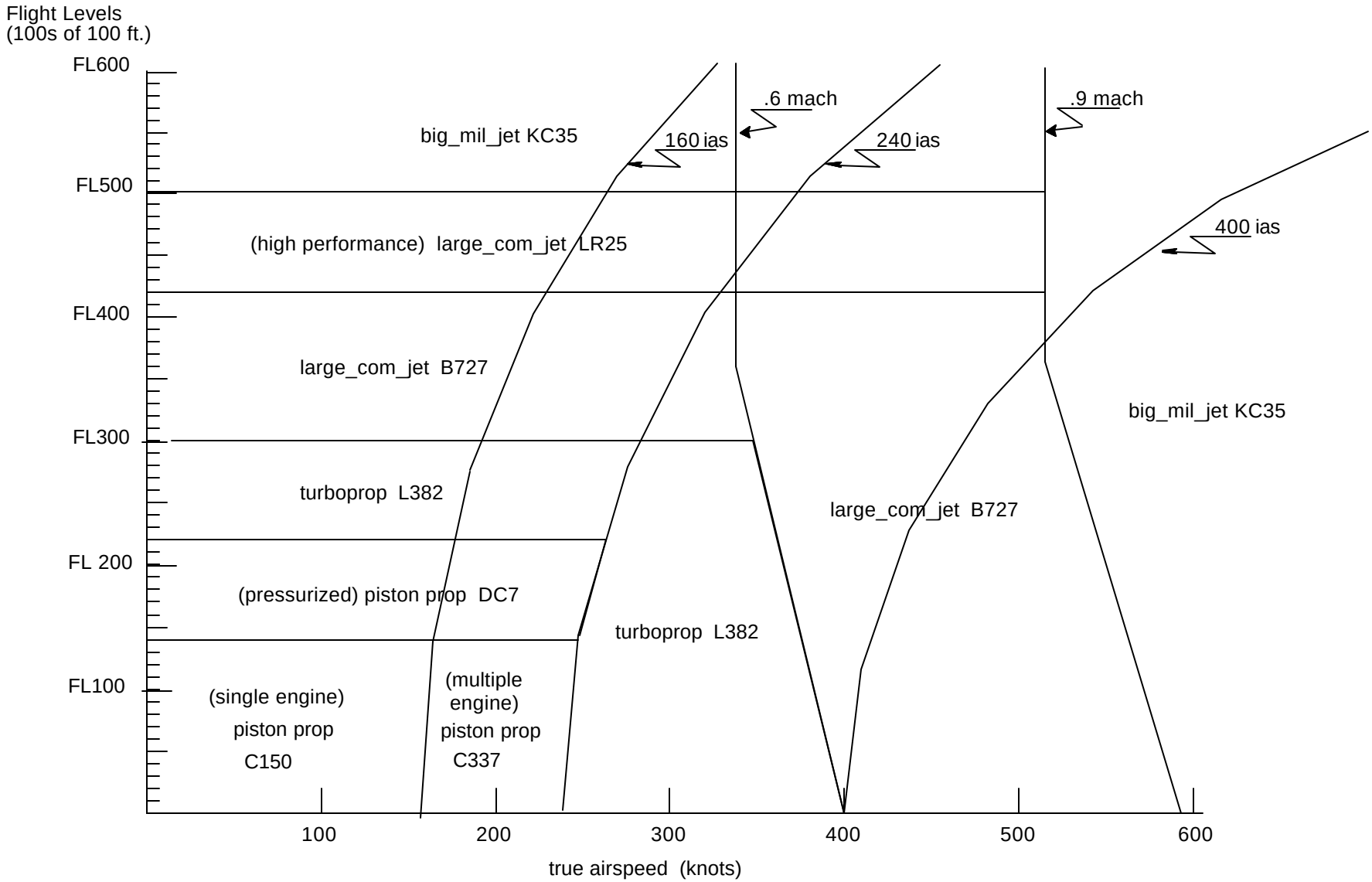


Figure 24-22. "Best Guess" Procedure for Assigning an Aircraft Template

There are nine descent profiles; each is determined by the aircraft's **grp** category/mach number range combination. The mach number is strictly dependent on the cruising altitude and speed and is computed by the *mach_vs_alt* procedure.

If the flight's descent is to be modeled, the distance along the flight path from cruise altitude to touch-down (i.e., **dist_descent**) must be computed. *Assign_a_profile* calculates this parameter by computing a cruising altitude index and using that to look up the **dist_descent** value from the **descent_by_alt** map. If the flight descent is not to be modeled, the **dist_descent** value is set to zero.

Check for Altitude/Distance Incompatibility. If a flight's take-off and/or descent is to be modeled, it is possible that, due to a short total distance, the aircraft's filed cruising altitude may be too high according to the profiles. If such an inconsistency is detected, the filed cruising altitude is decremented iteratively until the constraints between the total flight distance, the distance to the leveling out point (i.e., **dist_cruz**), and the distance down to the runway (i.e., **dist_descent**) are satisfied.

General Error Handling and Output: the Profile.Txt File. Error detection, assessment, and reporting is a primary function of *assign_a_profile.* There are currently eleven error types associated with the ascent and descent stages of the flight. There are two levels of severity in error assessment: minor inconsistencies which may or may not be adjusted, and severe errors. A flight with a severe error is no longer processed. The error types according to severity are described in Table 24-13. All errors regardless of severity are reported in an ASCII file named **Profile.Txt**. The file is opened and closed daily by the *parser*.

Table 24-13. Errors Associated with assign_a_profile

Minor Inconsistencies	Severe Errors
designator is not in the aircraft_descriptor map	cruise altitude is above FL600
cruise altitude is above aircraft ceiling	cruise altitude is zero or negative
cruise speed is too high for its grp category	flt length is negative
cruise speed is too high for flight below FL100	cruise speed too high for the database
cruise altitude vs. flt length conflict	cruise speed is zero or negative
	flight length is too long

24.5.5 The cleanup_evlist Routine

Purpose

Procedure *cleanup_evlist* performs post-processing on the **event list**. It examines the **event list** and removes various undesirable or erroneous features that are by-products of the route processing described in Section 24.4.3.

Input

Cleanup_evlist receives two types of input:

(1) Event list

(2) Flag indicating whether this is a tailored route or not

Output

Cleanup_evlist produces two types of output as well:

(1) "cleaned up" version of the event list

(2) Flags indicating the changes made and problems found in the original **event list**. The *Parser* takes these flags and keeps running counts which are displayed periodically

Processing

The cleaning up of the **event list** is performed in two steps. First, the cleanup procedure makes a series of checks in order to identify which events, if any, should be deleted. These events are *logically* deleted through the use of a Boolean tag array, which is the same size as the **event list** array. Logically deleting an event consists of setting the corresponding slot in the tag array to **false** and adjusting the **distance** and **heading** fields of the succeeding event. Second, events which have been logically deleted are *physically* deleted from the array simply by moving up all elements that are to be retained.

The different checks performed on the **event list** are described in the following sections.

Change en Route Airport Events into Unnamed Fix Events. Airports should appear only at the beginning or end of an **event list**. If an airport appear elsewhere, its type is changed to **unnamed_fix** and the **airport in middle of event list** flag is set.

Perform "Reasonableness" Check. To find stray points, the general heading of the flight is derived from the coordinates of the first and last point in the list. Each event with a distance of more than 500 miles is found. If the heading of one of these events is more than 45_ off from the general heading, and if the distance to the next event is over 500 miles, then this is considered an unreasonable sequence, and the **unreasonable event list** flag is set.

The flights most commonly flagged by this code are military and overseas flights. These flights seem to be the most erratic. The *fix_verify* routine does similar checks, but tosses fixes if they are out of range. The *fix_verify* routine code also uses more lenient bearing and distance checks.

Such **event lists** are not deleted or modified. The purpose of this flag is to allow us to track down problems in **event lists**, such as the non-unique fixes mentioned above.

Ensure that Sector, ARTCC, and Route Entries and Exits Match up. The **event list** is traversed to ensure that for each sector, ARTCC, and route entry there is a matching exit. Any extraneous or non-matching entries and exits are deleted unless it is the last ARTCC or route entry in the flight. Another exception to this is a tailored route. The event list in this case will begin at some point *during* the flight, not at take-off; thus, the first sector event could be an exit

rather than an entry. Such sector exits are not deleted. When sector events are deleted, one or more of the following flags are set:

- sector exit w/o matching sector entry
- sector entry w/o matching sector exit
- sector entry/exit pairs interleaved

Remove Sector Crossings Where Distance Travelled is Small. Since sector boundaries are represented not as straight lines, but as sequences of grid cells, they tend to be jagged. Should a flight path be along such a boundary, as shown in Figure 24-23, the corresponding **event list** may model the flight as continuously moving back and forth between the two sectors. This apparent zigzagging effect is marked by sector entries and exits that are only a few miles apart.

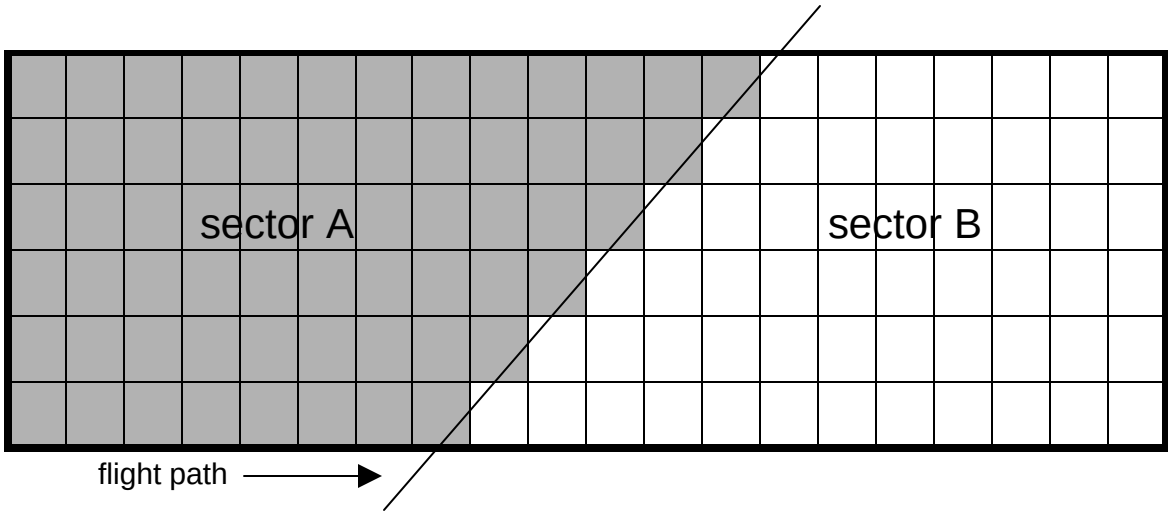


Figure 24-23. Extraneous Sector Crossings

To deal with this problem, the **event list** post-processor searches the list for sector entries that are separated from their matching sector exits by less than ten miles. All such pairs are deleted. The result will be an **event list** that shows the plane as having been in one sector or the other for the entire distance. This check also removes cases where a flight passes through the corner of a sector for a distance of less than ten miles.

Remove Fix Events Which Have a Very Low Altitude. Occasionally, fix events with zero altitudes will slip into the event list just prior to the final (airport) event. This happens when the *Route Processor* generates events for fixes that share the grid cell containing the destination airport. Such fixes are typically modeled to have an altitude of zero. Since these fixes are just a useless remnant of the data representation, they are removed. *Cleanup_evlist* searches the event list for all fixes with zero altitude, deletes them, and sets the **fix at very low altitude** flag.

Ensure That Event List End Points are Marked as Waypoints. The *ASD* draws the route represented by the event list by connecting each waypoint in the list. Therefore, in order to be sure that the entire route will be drawn, the first and last events should always have their

waypoint flags set. *Cleanup_evlist* sets these flags, if they were not set already.

Error Conditions and Handling

The *Parser* performs two levels of error-handling. Some errors may necessitate the termination and restarting of the process; however, most errors which occur are minor data extraction errors that do not terminally affect the parser process and may or may not affect the data sent to the *Flight Database Processor*.

Terminal errors are usually caused by failures within other processes or by network problems (e.g., stack and queues not being created or a mailbox not being found across the network). When a terminal error occurs, the process sends some diagnostic message to the screen and/or an error file and exits. The *Parser* is then restarted by *Nodescan* and reconnected to the other processes.

The *Parser* records any error occurring during the data extraction process in a time-stamped diagnostics file, which includes the NAS message text and a brief description of what caused the error. Errors occurring during the creation or cleaning up of the event list are catalogued by the *Parser*'s assignment of certain error values to unsuccessful event list operations. The *Parser* checks the return status of the creation or cleaning-up function and, if it finds an error, logs an appropriate message in the diagnostics file. Unless the *Parser* finds virtually no usable data in a NAS message, the data extracted are shipped to the *Flight Database Processor*, the idea being that receiving partial data for a flight is better than receiving no data for a flight.

24.6 Parser Source Code Organization

The Parser source code resides in C files under configuration management using ClearCase.

24.7 Parser Data Structure Tables

24.7.1 The nas_msg_type Data Structure

The **nas_msg_type** data structure is used to pass necessary flight information between the *Parser* process and the *Route Processor* function when a flight path route (field 10) is present. **Nas_msg_type** is a record structure comprised of all the data fields needed to create an event list from the flight's route. This flight information is used by the *Route Processor* function to create an event list that models the aircraft's route of flight. The completed event list is then passed back to the *Parser* process. Table 24-14 illustrates the structure.

24.7.2 The map_data_type and no_route_data_type Data Structures

The **map_data_type** data structure is used by all the main routines of the *Parser* process. It is a record structure comprised of all the possible data fields that may be extracted from any input message type. It is used to store flight information culled from a single input message. When all the flight data from a message has been extracted and converted, it is passed to the *Flight Database Processor* (via the *parser.relay* process) using this data structure. See Table 24-15 for details on the **map_data_type** structure

For messages that have no field 10s (TZs, DZs, AZs, and RZs), the information is passed to the *Flight Database Processor* through the **no_route_data_type** structure (see Table 24-16).

Table 24-14. nas_msg_type Data Structure

nas_msg_type				
Library Name: ttm_openlib		**Purpose:** To contain specific flight information needed to create an event list from the given route (field 10) by the route processor.		
Element Name: parser_routeproc.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_id	NAS flight identifier	1 or more letters followed by	--	string10
aircraft_type	NAS aircraft type	Letter fol. by up to 3 letters &	--	string4
dep_time	Flight's departure time.	Minutes since midnight	0 – 1439	short
cruising_velocity	Filed cruising speed.	(nautical miles * 100) per minute.	--	short
cruising_altitude	Filed cruising altitude.	Hundreds of feet.	--	short
message_type	Type of NAS message currently being parsed.	Enumerated type.	1 – 14 +	short
new_data	Set if altitude and speed are modified by profile a/c	————————	T/F	boolean
field_10	Text of route to be	————————	--	array [1…300] of char
time_type	Prefix of coordination time	————————	--	char
n_ti_fix	Coordination fix from NAS mes-sage.	Lat/lon or airport name or fix	--	string14

\+ 1 = AF (amendment message)

2 = AZ (arrival message)

3 = CT (estimated departure clearance time

4 = DZ (departure message)

5 = FZ (flight plan message)

6 = RZ (cancellation message)

7 = UZ (update message)

9 = TZ (5-minute location update)

10 = FA (feedback loop type;sends messages from FDB back to Parser)

11 = CCC (ARTCC status message)

12 = SYNC (updated ASD message)

13 = FS (FZ from schedule database)

14 = RS (RZ from schedule database)

15 = EDCT (controlled departure time message from ETMS)

16 = TO (oceanic position updates)

22 = TA (global position updates – not currently being received)

Table 24-15. map_data_type Data Structure

map_data_type				
Library Name: parserfdb_openlib **Element Name:** message_structsh		**Purpose:** Contains data extracted from a NAS message with a route (field 10) to be passed to the FDBP. Not all fields will be filled depending on the message type.		
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
message_type	specific type of NAS message			short
message_time_sta	encoded time of when message was received.			CALCLOCK
center_origin_of_ message	ARTCC that message came from			char
flight_id	unique flight identifier			string7
computer_id	3-digit flight identifier within an ARTCC		000 – 999	string3
num_aircraft	number of aircraft			char
ac_eqp_prefix	aircraft equipment prefix indicator			char
aircraft_type	type of aircraft			string4
ac_eqp_suffix	aircraft equipment suffix indicator			char
user_category	flight usage			usercat_t
cat_class	flight class			flight_regs_t
actype_class	aircraft class			actype_t
ac_category	aircraft category			ac_cat_t
ac_weight_class	aircraft weight class			ac_weight_t
speed	flight's speed			short
speed_type	type of speed			char

Table 24-15. map_data_type Data Structure (continued)

map_data_type (continued)				
Library Name: parserfdb_openlib **Element Name:** message_structsh	**Purpose:** Contains data extracted from a NAS message with a route (field 10) to be passed to the FDBP. Not all fields will be filled depending on the message type.			
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
coordination_fix_event_2	TO message. The 2nd predicted position			erect
arrival_fix_event	event created at arrival fix			short_erect
dep_center	departure airport			char
arr_cen	arrival airport			char
tail_id				string7
actual_position_time	an actual position time: hhmmss			short_time_t
wind_direction	wind direction in degrees, measured clockwise off North	degrees		short
wind_speed	wind speed in knots	knots		short
air temperature	air temperature in degrees centigrade	degrees centigrade		short
az_source	(future use)			char
geographic_flags	(future use)			short
pad	padding for alignment: unused			string8
af_field_changes	field numbers changed by AF message	NAS message field	10	short
new_flight_id	(future use)			string7
dsg_index				short
acft_index	index of aircraft type			short
ascent_index	index of ascent table			short
descent_index	index of descent table			short

Table 24-16. no_route_data_type Data Structure

no_route_data_type				
Library Name: parserfdb_openlib		**Purpose:** Contains data extracted from a NAS message without a route to be passed to the FDBP. Not all fields will be filled depending on the message type.		
Element Name: message_structsh				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
message_type	specific type of NAS message			short
message_time_stamp	encoded time of when message was received.			CALCLOCK
center_origin_of_message	ARTCC that message came from			char
flight_id	unique flight identifier			string7
computer_id	3-digit flight identifier within an ARTCC			string3
num_aircraft	number of aircraft			char
ac_eqp_prefix	aircraft equipment prefix indicator			char
acft_type	type of aircraft			string4
ac_eqp_suffix	aircraft equipment suffix indicator			char
user_category	flight usage			usercat_t
cat_class	flight class			flight_regs_t
actype_class	aircraft class			actype_t
ac_category	aircraft category			ac_cat_t
ac_weight_class	aircraft weight class			ac_weight_t
speed	flight's speed			short
speed_type	type of speed			char
dept_point	flight's departure airport			string4
altitude1	flight's altitude	altitude / 100		short
altitude2	flight's altitude	altitude / 100		short
altitude_type	flight's altitude type			char

Table 24-16. no_route_data_type Data Structure (continued)

no_route_data_type (continued)				
Library Name: parserfdb_openlib **Element Name:** message_structsh	**Purpose:** Contains data extracted from a NAS message without a route to be passed to the FDBP. Not all fields will be filled depending on the message type.			
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
dest_point	arrival airport			string4
est_time_arr	estimated time of arrival			short
coordination_fix_eve	coordination fix event			erect
coordination_fix_eve	TO message. The 1st predicted position			erect
coordination_fix_eve	TO message. The 2nd predicted position			erect
arrival_fix_event				short_erect
dep_center	departure airport center			char
arr_cen	arrival airport center			char
tail_id				string7
actual_position_time	an actual flight position time hhmmss			short_time_t
wind_direction	wind direction in degrees, measured clockwise off North			short
wind_speed	wind speed in knots			short
air temperature	air temperature (centigrade)			short
az_source	(future use)			char
geographic_flags	(future use)			short